# Appendix I

## A population genetics pedigree perspective on the transmission of *Helicobacter pylori*

"Support bacteria. It's the only culture some people have."

unknown

# A population genetics pedigree perspective on the transmission of *Helicobacter pylori*

Wayne Delport[1], Michael Cunningham[1], Brenda Olivier[2], Oliver Preisig[3] & Schalk W van der Merwe[2]

[1]Molecular Ecology and Evolution Programme, Department of Genetics, University of Pretoria, Pretoria, 0002, South Africa

[2]Hepatology/GI-research laboratory, Department of Internal Medicine, University of Pretoria, Pretoria, 0002, South Africa

[3]Inqaba Biotech, Pretoria, South Africa

Corresponding Author:

Dr Schalk W van der Merwe
Department of Internal Medicine and Gastroenterology and GI/Hepatology research laboratory
Lab 2.75, Pathology Building, Basic Medical Sciences Campus
University of Pretoria,
South Africa
Email: svdm@doctors.netcare.co.za
Telephone: +27 12 6640187
Fax: +27 12 6648167

**Abstract**

The inference of transmission pathways for medicinally important bacteria is crucially important to our understanding of pathogens and the design of efficient pathogen control measures. Here we report an analysis of transmission in *Helicobacter pylori*, a major carcinogen, based on simulations of nucleotide sequences across extended familial pedigrees. Previous epidemiological studies of this organism have been characterized by methodological inadequacies, related to the chosen study population, familial samples sizes, and means of inference. The approach followed here offers improvements in the inference of transmission, including (i) the study of a community with a high prevalence of *H. pylori*, (ii) detailed family pedigrees spanning three generations, (iii) high-resolution analysis of nucleotide sequences, and (iv) a model that incorporates the occurrence of mutation and recombination between the times of infection and subsequent sampling for genetic analysis. Contrary to previous studies, our results demonstrate that the transmission of *H. pylori* is predominantly non-familial, with only a low proportion of mother-to-child transmission events. These results are interesting from both a medical and an evolutionary standpoint. In the former, efficient control measures and beliefs about the sources of *H. pylori* infection should be re-evaluated. Evolutionarily, our results contradict the hypothesis of strict vertical transmission, which has been presented as explanation for the strong correlation between human population history and *H. pylori* diversity. Thus the paradox of persistent phylogenetic structure, despite a permissive mode of transmission and extremely high recombination rates, must be solved elsewhere. Here we consider the potential for recombination events to maintain genetic structure in light of horizontal transmission.

## Introduction

*Helicobacter pylori* is a gram-negative bacterium that colonizes the gastric mucosa (1). This infection is associated with chronic gastritis, peptic ulcer disease, MALT lymphoma and gastric adenocarcinoma, has a major impact on public health (2), and has been classified as a group I carcinogen by the International Agency for Research on Cancer (3). Although much is known of the virulence of *H. pylori* (4), the potential transmission pathways for the bacterium are unresolved (5,6). Potential transmission routes include oral-oral and fecal-oral, both with and without intermediate transmission steps (7). These transmission pathways are supported by a plethora of studies demonstrating the presence of *H. pylori* isolates, either by culturing or PCR techniques, both in the mouth environment (see 5 for a review), and in stool samples (8,9,10). Further clues regarding transmission lie in the fact that infection is more prevalent in developing countries, where it can assume epidemic proportions with up to 80% of the population being infected (11). Such observations suggest that *H. pylori* infection may be associated with low socio-economic status, and overcrowded living conditions, as has been suggested previously (12,13). Transmission studies in developed countries suggest a person-to-person mode of transmission (12). Furthermore, epidemiological and DNA-based studies have suggested that a parent-offspring transmission pathway, mostly mother-to-child, is responsible for transmission within the family and that infection outside the family rarely occurs (12,14,15,16,17).

Several methodological problems are consistent across these studies, and can be summarized as (i) a low incidence of infection within the study population, related to its socio-economic status (16,18), (ii) small sample sizes within families (16,17), and (iii) inappropriate methods of inference for transmission (16,17,18). The first of these methodological problems relates to the observations that poor sanitation, overcrowding and generally low socio-economic status are factors that determine susceptibility to *H. pylori* infection (12,13,19). Given that developed countries have shown a decrease in *H. pylori* incidence without targeted intervention (20,21), the *status quo* of *H. pylori* and its transmission route should be addressed in developing countries where incidence is high, and the bacterium continues to present itself as a serious health concern. Secondly, few studies have compared *H.pylori* genotypes from both parents and several children within individual families (16,17). The observation of transmissions from a mother to each of her children should be the means by which the route of infection is evaluated such that one can distinguish socially mediated transmission from a strictly vertical pathway. Third, transmission studies have typically used presence/absence measures of *H. pylori* infection via $^{13}C$ Urea breath tests (16,18), presence/absence of antibodies (17,18), restriction enzyme digestion of amplified genes (22) or analyses of peptide sequences (23). Although, the latter two methods are an advance over $^{13}C$ Urea

breath tests as they allow comparison of *H. pylori* genotypes in parents and children, these methods do not provide the high resolution of nucleotide sequence based studies and are more susceptible to convergent changes. Furthermore, previous studies have not considered the potential for mutation, and/or recombination, to occur between the times of transmission and sampling. Understanding the transmission process is an essential step towards controlling the spread of *H. pylori*. In this manuscript we derive DNA sequence data from extended family pedigrees in a high prevalence community, and use population genetics tools and computer simulations to infer the transmission patterns of *H. pylori*. We show that transmission in this population is not predominantly vertical, and that horizontal transmission from the community plays a major role in the spread of *H. pylori*.

## Methods

### *Sampling and gene sequencing*

The study population comprised 105 healthy individuals from a rural, South African, black community (Ogies, Mpumulanga) that have been followed as part of a long-term surveillance program on the epidemiology of *H. pylori* (6,7). This population had many of the risk factors that are associated with a high prevalence of *H. pylori* infection (12, 13). Ethical approval for the current study was obtained from the University of Pretoria and the Hospital Review board of the Unitas hospital. Endoscopy was performed in 90 individuals. DNA was isolated directly from gastric biopsy samples and fragments from three housekeeping genes (UreI, UreC & MutY, Figure 1) were PCR amplified and sequenced. Direct DNA extraction of biopsy samples was preferred over culturing since the latter may result in *in vitro* sequence change, and may also reduce genotypic diversity within a sample. Where multiple genotypes were detected in a single biopsy sample, we cloned PCR products and sequenced a selection of these cloned fragments to identify unique genotypes. Further details of gene sequencing are available in the online supporting material on the PNAS website.

### *Sequence analysis*

Sequences, from the three genes, were analysed in order to identify the predominant route of transmission of *H. pylori.* In summary, (i) neighbour-joining phylograms were used to represent phylogenetic structure of *H. pylori* within the community, (ii) the Structure 2 software (24) was used to align within-community genetic diversity sampled in this study with that found in global *H. pylori* samples, and (iii) statistical comparisons and a custom simulation model were developed to make inferences regarding the most likely path of transmission (Table 1). We simulated transmission of *H. pylori* by drawing from the available sequences in the pedigree and from the community (Figure 2). The source for each transmission event was determined by a random draw

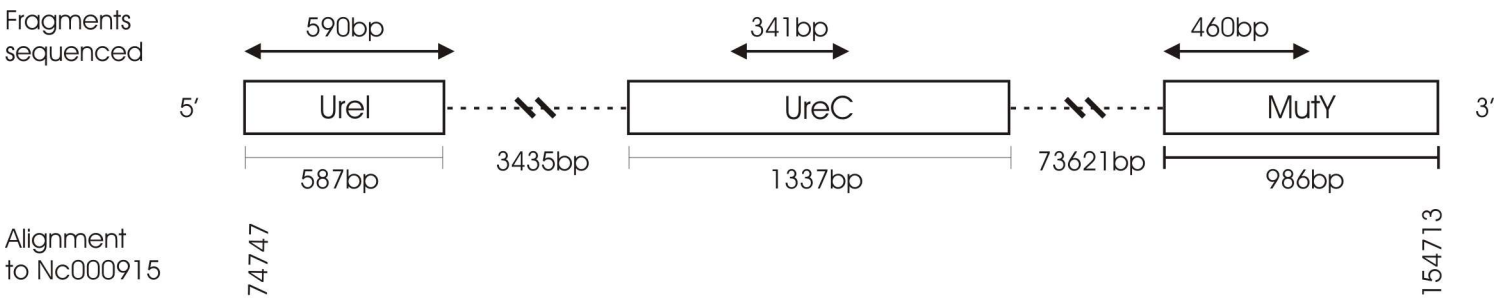


**Figure 1**: Location of genes sequenced in this study on the *H. pylori* genome. Alignment to the complete *H. pylori* genome sequence on Genbank (NC000915) is shown, as are lengths of fragments sequenced and inter-gene distances in base pairs.

**Table 1:** Alternative scenarios assessed in the pedigree based simulation model. Each row indicates a transmission scenario, giving rates of infection from different sources. Transmission hypotheses range from predominantly vertical (models 1, 2) to predominantly horizontal transmission, with infection acquired either within families (model 6) or externally, from the wider community (model 7). Further details of the transmission model are available as supporting material on the PNAS website.

| | **Vertical** | | **Horizontal** | | |
|---|---|---|---|---|---|
| | **maternal** | **paternal** | **sibling** | **household** | **community** |
| 1 | 0.900 | 0.050 | 0.000 | 0.000 | 0.050 |
| 2 | 0.475 | 0.475 | 0.000 | 0.000 | 0.050 |
| 3 | 0.450 | 0.050 | 0.000 | 0.000 | 0.500 |
| 4 | 0.250 | 0.250 | 0.000 | 0.000 | 0.500 |
| 5 | 0.250 | 0.250 | 0.167 | 0.167 | 0.167 |
| 6 | 0.025 | 0.025 | 0.500 | 0.400 | 0.050 |
| 7 | 0.025 | 0.025 | 0.050 | 0.050 | 0.850 |

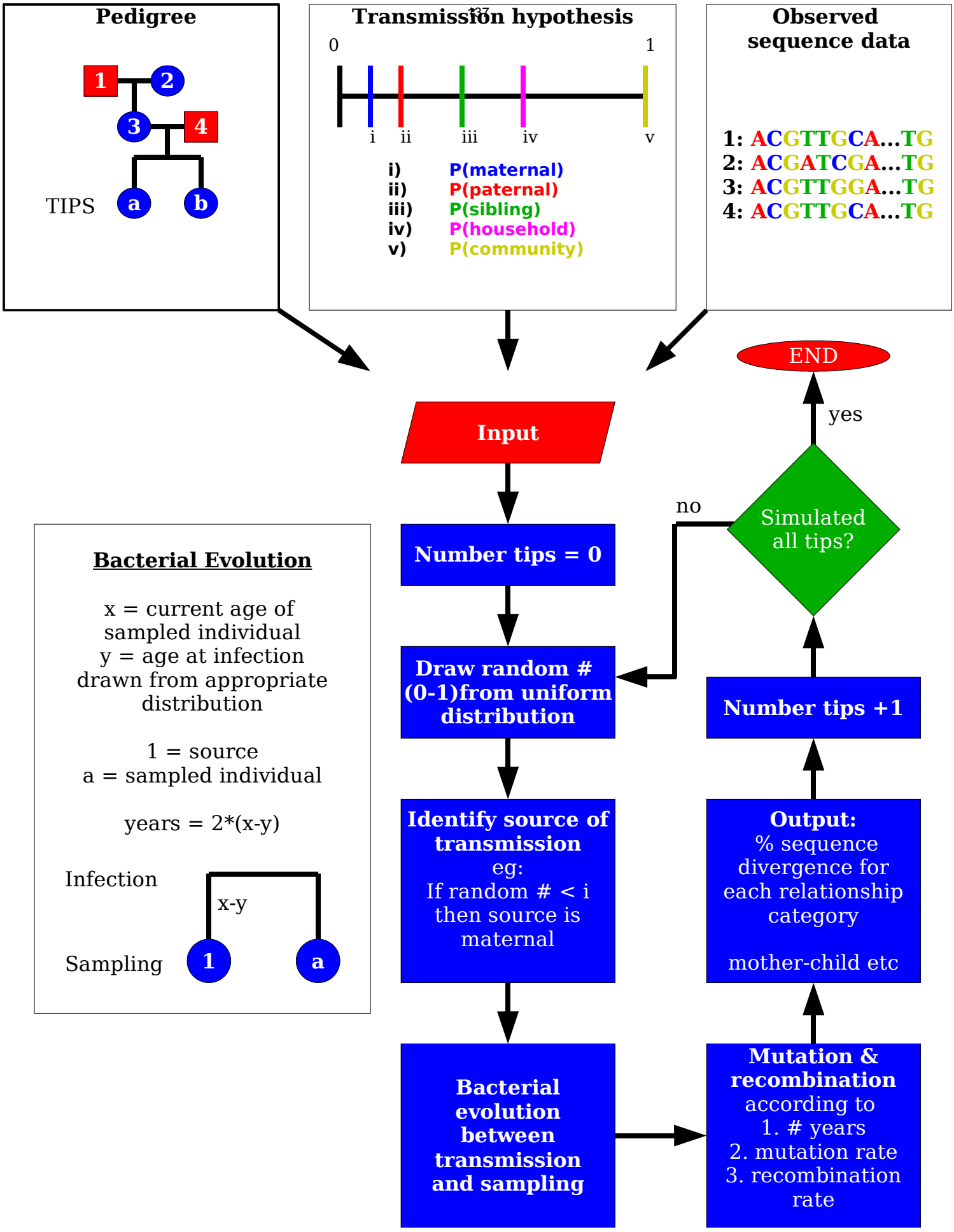


**Figure 2**: Schematic of the transmission simulation model used to make inferences of the most likely path of transmission of *H. pylori*. Note only a framework of the model is provided. A detailed description is available as online supporting material.

given the transmission hypothesis (Table 1). Simulations were repeated 1000 times for each of these scenarios, to assess general trends in pairwise-sequence comparisons under each transmission hypothesis. The details of these analyses, and a complete description of the simulation model, are published as online supporting information on the PNAS website.

**Results**

*Sequence Analysis*

Seventy five individuals (83%) were *H. pylori* positive by PCR and histology. Multiple genotypes were detected in ten (13%) of the individuals sequenced. Co-infection with multiple *H. pylori* genotypes has been suggested in some previous studies (25, 26), although most studies have reported a single genotype per individual. The low level of multiple infection detected in this study, is unlikely to influence the inference of transmission and furthermore, cloned sequences from individuals with multiple genotypes were similar to those from direct sequencing. Consequently, all subsequent analyses were performed assuming one strain per individual. Preliminary sequence analyses show high levels of gene diversity for each of the genes sequenced (Table 2). In general nucleotide diversity is higher for MutY, than for either of the other genes (UreI and UreC). There are low levels of coding substitutions relative to silent substitutions, and thus there is no evidence for selection among the sequenced genotypes at these genes. Given that the fragments sequenced are from general *H. pylori* housekeeping genes this result is expected. Finally, phylogenetic conflict between adjacent segregating sites suggest a minimum estimate of between 23 and 37 recombination events within this sample for each of the three genes.

*Global H. pylori diversity and population structure*

The purpose of the Structure analysis (see supporting materials and methods) was to place individuals from this study within the global context of previously described *H. pylori* population groups (30). Modern *H. pylori* population groups include hpAfrica1, hpAfrica2, hpEurope and hpEastAsia (30). First, we tested the assignment power of the three genes sequenced in this study, versus eight genes in the former study (30). Two of the three genes sequenced here, MutY and UreI, were common to both studies but UreC was not sequenced in the earlier study. Assignment of global *H. pylori* sequences to the above groups was identical to that in the former study, irrespective of whether UreC was included as missing data, or excluded, and thus provides support for the discriminatory power of our three gene data set. This probably reflects limited recombination between population groups among the fragments sequenced here (Figure 2 in 30), and the exclusion of the largely admixed European samples (see online supporting materials).

**Table 2:** *H. pylori* diversity indices at three independent loci with mean (and standard deviations) calculated in DnaSp (31). Gene diversity is the probability that two randomly-chosen isolates differ at that locus. Nucleotide diversity is the probability that two selected isolates differ at a randomly chosen nucleotide site. Segregating sites are the number of nucleotide positions that differ among isolates. $K_a$ is the number of coding substitutions per coding site, and $K_s$ is the number of silent substitutions per silent site. Ratios of $K_a/K_s > 1$ indicate selection on the coding sequence of a locus. $R_e$ is the minimum number of recombination events as estimated using the four-gamete test (32) in DnaSP (31).

| | ***UreI*** | ***UreC*** | ***MutY*** |
|---|---|---|---|
| Sequence Length (bp) | 590 | 341 | 460 |
| Individuals (n) | 80 | 79 | 79 |
| Gene diversity | 0,992 (0,003) | 0,998 (0.004) | 0,990 (0.004) |
| Nucleotide diversity | 0,041 (0,001) | 0,050 (0,002) | 0,068 (0,002) |
| Segregating sites | 89 | 71 | 105 |
| $K_a/K_s$ | 0,045 (0,041) | 0,050 (0,060) | 0,078 (0.040) |
| Min. Recombination ($R_e$) | 26 | 23 | 37 |

Secondly we assigned *H. pylori* genotypes from the Ogies community to the previously recognized global population groups. This was performed with a no-admixture model, and with a linkage and admixture model (see online supporting material for details). In the no-admixture model, 47 isolates were assigned to hpAfrica1, 23 to hpAfrica2 and 2 to hpEastAsia. Seventy-two of the seventy-four Ogies individuals were assigned to identical clusters when either two genes (UreI and MutY), or three genes (UreI, UreC, MutY) were used. The incorrect assignments occurred with isolates 112 and 189, which were assigned to hpAfrica1 with three genes, and hpAfrica2 with two genes, for the former, and the converse for the latter. These isolates have contrasting positions in single gene phylograms and their varying assignment to either hpAfrica1 or hpAfrica2 clearly results from recombination events between UreI / UreC and MutY (Figure 3). The linkage and admixture model allows isolates to have mixed origins, and determines the proportion of each individual's nucleotides that are derived from each previously identified ancestral population (30). *H. pylori* nucleotide polymorphisms from the Ogies community were assigned to the ancestral Africa1, ancestral Africa2, ancestral Europe1 and ancestral Europe2 populations (Figure 4). In most cases the proportion of nucleotides derived from a single ancestral population was high. The nucleotide assignments indicate the presence of two clear groups within the community, ancestral Africa1 and ancestral Africa2, and some additional individuals of mixed ancestry, which appear to mostly comprise ancestral Europe2 (Figure 4). Despite the contribution of several population groups to genotype diversity in the Ogies community, within individual ancestral population diversity is lower in this single cohesive population than in the global *H. pylori* sample (Figure 4).

*Within-community population structure*

Further analyses using the Structure software (24) were conducted to determine the within-community population structure of *H. pylori,* and to estimate recombination parameters. These analyses, using mean likelihoods over five independent runs, estimated the number of subpopulations, K, to be either four or five, as was previously determined for the global *H. pylori* data (30). Some independent replications supported six or seven subpopulations but the inferred likelihoods for these values were not consistently higher (Table 4 published as online supporting material). The admixture model with correlated allele frequencies can tend to overestimate the number of inferred subpopulations as it permits the existence of subpopulations with similar allele frequencies (31). Most nucleotide polymorphisms in the Ogies community are derived from four ancestral populations (Africa1, Africa2, Europe1 and Europe2), with a few admixed individuals, carrying a small proportion of ancestral EastAsia nucleotides.

The Structure software allows the estimation of recombination rate, *r,* and the mean size of recombined sequence chunks. The parameter *r* was calculated, from an admixture model with four

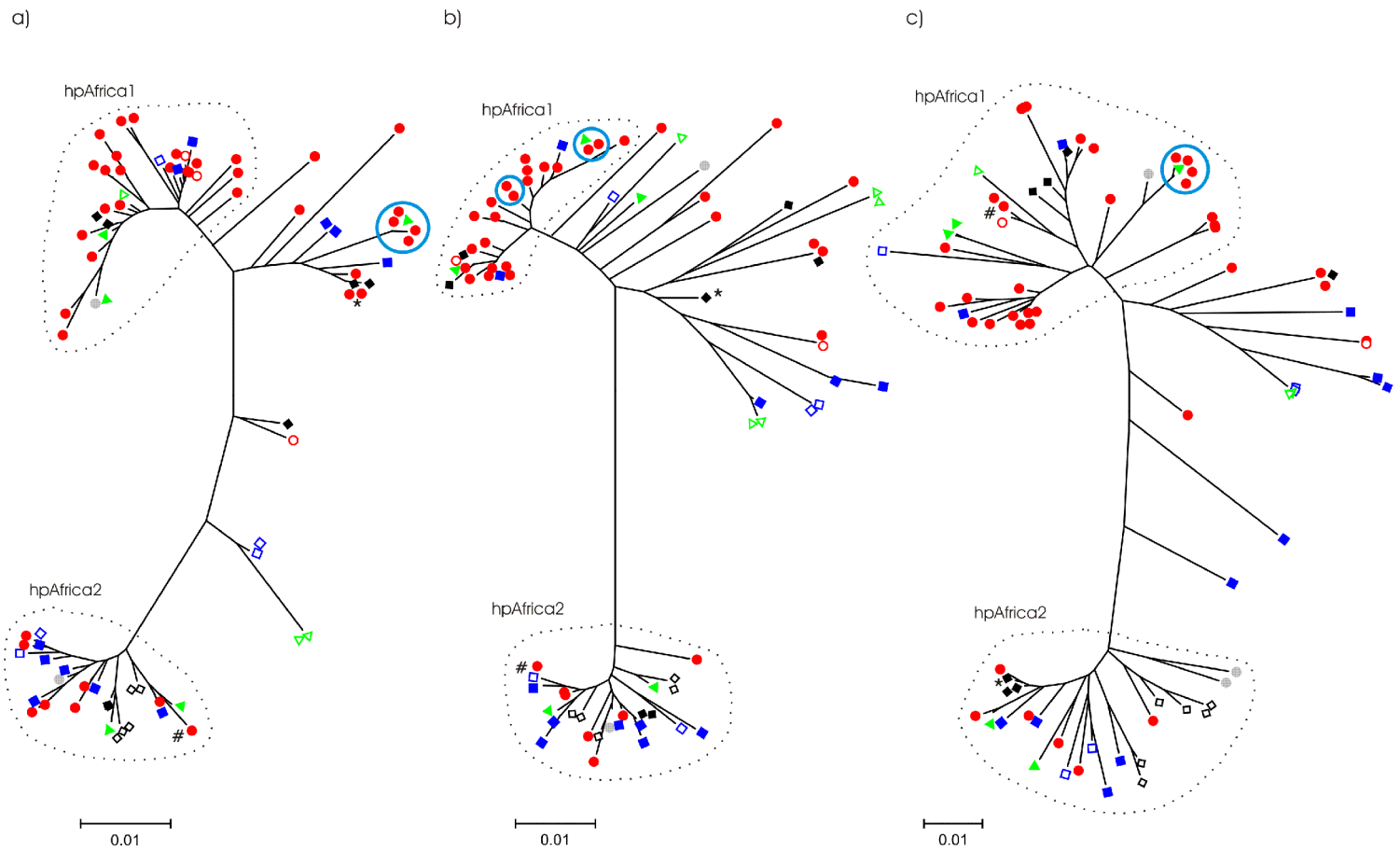


**Figure 3**: Neighbour-joining phylograms derived from three genes sequenced in this study: a) UreI, b) UreC and c) MutY. The three phylogenetic 'groups' used for the creation of new alleles in the simulation broadly correspond to hpAfrica1, hpAfrica2 and the complement of these. Pedigree information is indicated by symbols, where filled red circles = family 12, filled blue squares = family 13, filled green triangles = family 21, filled black diamonds = family 39, open red circles = family 48, filled gray circles = family 49, open blue squares = family 50, open green triangles = family 51, open black diamonds = family 52. * and # indicate the alternate placements of individuals 112 and 189, respectively.

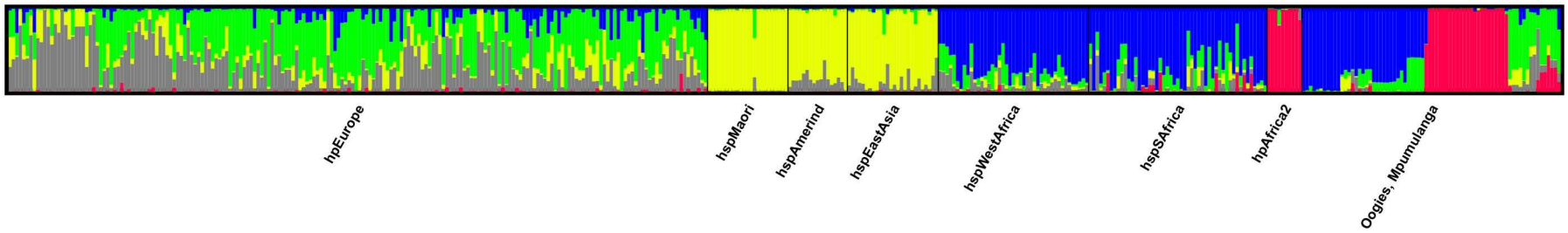


**Figure 4**: Proportion of nucleotides derived from each of the ancestral populations previously identified (31), for global samples and for within-community samples considered in this study. The proportion of nucleotides was estimated with for three genes, UreI, UreC and MutY, using a linkage and admixture model in Structure 2 (22). UreC was specified as missing data, since this gene was not sequenced in previous studies. Each bar shows the proportion of nucleotides derived from each of five ancestral populations, where colours are as follows: blue: ancestral Africa1, red: ancestral Africa2, gray: ancestral Europe1, green: ancestral Europe2, yellow: ancestral East Asia. The figure was plotted using Distruct (48).

subpopulations, as 0.0007 ± 0.0001 recombination events per nucleotide per year. This estimate is considerably less than the 0.0296 estimated previously (30). This difference implies that either the time since admixture is shorter, or the rate of recombination per nucleotide is lower in our data set. Furthermore, the estimated value of *r* implies an average chunk size of 1428bp versus that of 34bp calculated from the global data set (30). This is consistent with the observation above that within individual levels of admixture are lower in this single community than in the global sample, despite similar levels of genotype diversity.

*Transmission analysis*

Transmission of *H. pylori* was inferred using two approaches. The first approach used categorical tests to determine whether the number of pairwise comparisons between individuals carrying identical or similar sequences, within a particular relationship category, differed from that expected under a random assignment of genotypes to individuals within the sample. We compared genotypes within mother-child, parent-offspring, sibling, extended family, housemate and spouse relationship classes. To test for associations between genotypes within relationship classes we used Chi-square values, with the test distribution for these pairwise comparisons generated from 1000 random permutations of genotype assignments (Table 3). Results from these Chi-square permutation tests indicate that (i) parents are significantly more likely to share similar *H. pylori* genotypes with their children than are unrelated individuals, (ii) siblings are also likely to share *H. pylori* genotypes, (iii) individuals from the same household show the highest frequency of genotype sharing (evident as large Chi-square values), irrespective of their family relationships, and (iv) spousal partners are no more likely to share *H. pylori* genotypes with each other than with anyone else in the community. These results are consistent with some degree of transmission through childhood social interaction. Most individuals, however, carry substantially different *H. pylori* genotypes, irrespective of their relationships, which suggests that most *H. pylori* infections are acquired outside the family.

The second approach involved the construction of a probabilistic model that used pedigree information and sequence data derived from the study population to simulate transmission processes (Table 1). This simulation model is preferred over categorical tests as it incorporates mutations and recombination events that may have occurred since infection, and allows one to investigate patterns of genetic diversity occurring under multiple contrasting transmission pathways. As with all such models, however, it was first necessary to evaluate simulation results in terms of whether these were comparable with the sample, and whether these are sufficient to discriminate alternate transmission pathways. Since this model simulated a single transmission event for each of 26 individuals one would not expect summary statistics (gene diversity and nucleotide diversity) or phylogenetic structure of the simulated data to differ substantially under the

**Table 3:** Results of Chi-square permutation tests of association between similar genotypes and particular relationship categories. The test distribution was calculated from 1000 random permutations of the data matrix. Significance at the 0.05 level is indicated in bold.

| **Relationship** | **Chi-square** | | | ***P*** | | |
|---|---|---|---|---|---|---|
| | ***UreI*** | ***UreC*** | ***MutY*** | ***UreI*** | ***UreC*** | ***MutY*** |
| **Mother-child** | 6.9 | 3.7 | 10.4 | **0.008** | **0.037** | **<0.001** |
| **Parent-offspring** | 11.2 | 1.2 | 12.7 | **<0.001** | 0.208 | **0.001** |
| **Siblings** | 7.8 | 0.13 | 26.9 | **0.005** | 0.565 | **<0.001** |
| **Extended family** | 0.32 | 69.9 | 48.3 | 0.57 | **<0.001** | **<0.001** |
| **Housemates** | 33.1 | 17.5 | 96.5 | **<0.001** | **<0.001** | **<0.001** |
| **Spouses** | 0.7 | 0.7 | 0.2 | 0.068 | 0.37 | 0.199 |

alternate transmission hypotheses. Pairwise comparisons of parent-child divergence, however, should show marked differences, according to the frequency of vertical transmission in a particular transmission scenario. Mismatch distributions, distributions of gene diversity and nucleotide diversity and distributions of segregating sites for three contrasting transmission hypotheses (scenarios 1, 5 and 7 from Table 1) were comparable and similar to the observed data (Figure 6 published as online supporting material). Furthermore, permutation tests in PAUP*4b10 (35) using the observed topology (Figure 3) as a constraint, demonstrated that the simulated data was highly consistent with the observed phylogeny for each of these three alternate transmission hypotheses (in all cases $p = 1.0$). These results indicate that the simulation model does not perturb within-community phylogenetic structure (evident in Figure 3). To assess the potential of mother-child sequence divergence to discriminate alternate transmission hypotheses, we calculated confidence limits, using jackknife and bootstrap procedures, on distribution statistics derived from 1000 simulations. These results indicate that this approach is powerful, with narrow confidence limits on expected statistics (Table 5a published as online supporting material). In particular, there is a shift from a strongly right-skewed distribution of mother-child divergences ($g_1>0$), in vertical transmission models to a strongly left-skewed distribution ($g_1<0$), when infection is predominantly acquired horizontally and outside the family (Table 5a). These results confirm that average pairwise sequence divergences between mothers and their children are low under strict vertical transmission models and high under permissive horizontal transmission models.

In order to infer which of the transmission hypotheses were most likely to generate the observed data we conducted Kolmogorov-Smirnov tests comparing observed distributions of pairwise divergence statistics against those simulated under various transmission scenarios. The observed distribution of within-household sequence divergence values for the 26 focal individuals was significantly different from all the simulated transmission scenarios (Table 6 published as online supporting material). In contrast, the sequence divergence values among siblings were consistent with all the transmission models considered. These cases show rigorous and insufficient discrimination respectively. The inability to discriminate between hypotheses for among housemate comparisons may partly be due to the difficulty in distinguishing instances of vertical and horizontal transmission within households, where individuals can obtain identical genotypes both from specific parent-offspring interactions or from less specific social contacts. Father-child divergence distributions showed significant deviations from the observed data in five of the seven hypothesized transmission scenarios, including those with strictly vertical transmission (Table 6). Observed mother-child divergences were significantly different from both vertical transmission scenarios (Table 6), however elements of both parent-offspring and social transmission are evident in the observed data (Figure 5). In general, scenarios with a high probability of vertical

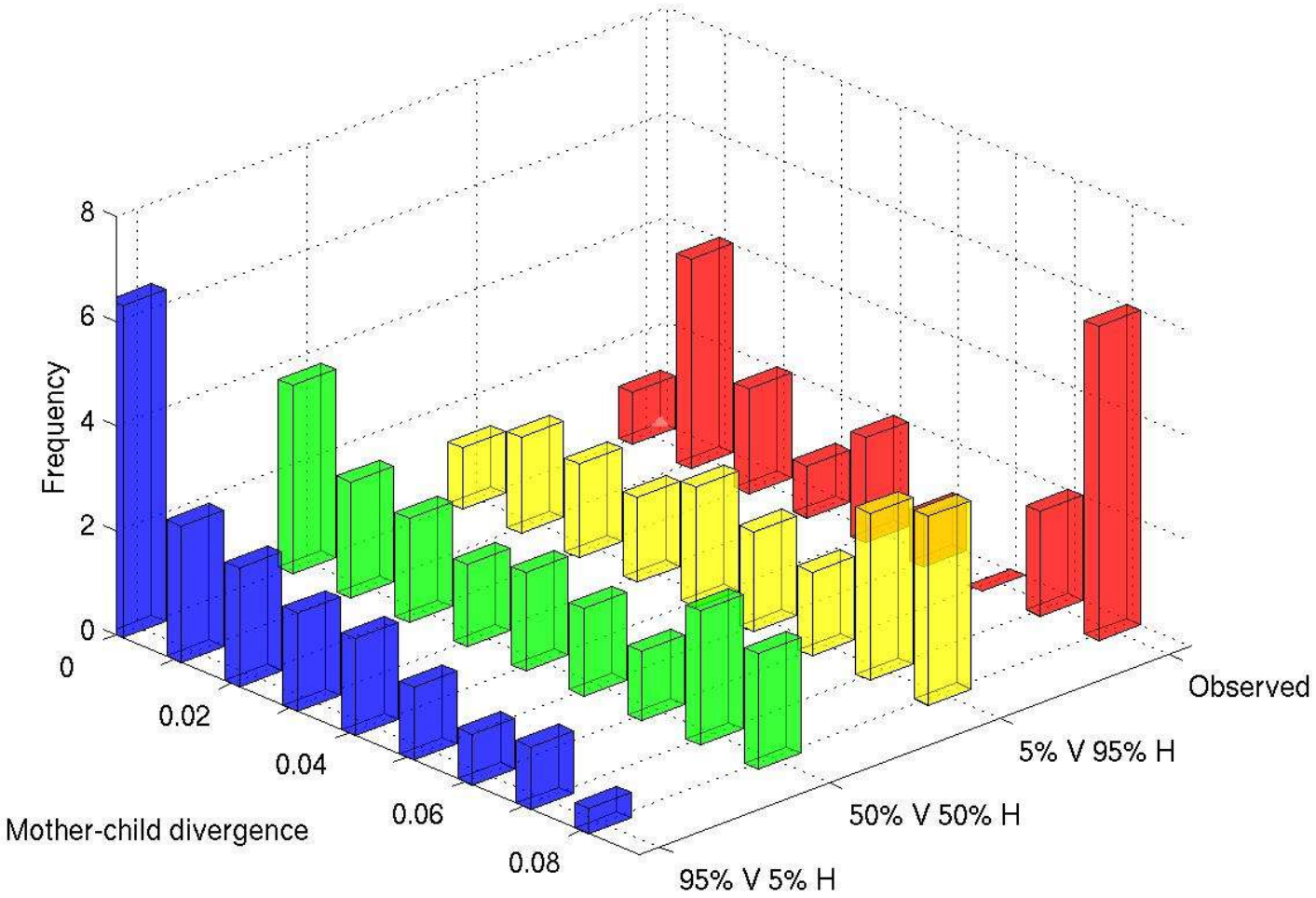


**Figure 5**: Distribution of mother-child sequence divergence for three simulated transmission hypotheses and the observed data. The mean of 1000 simulations is shown for each of the simulated data sets.

transmission, and scenarios with low probabilities of infection from the community were least consistent with the observed data (Table 6). Jackknife and bootstrap estimates of distribution statistics for the observed data have wider confidence limits than the simulated data (Table 5), as expected given the smaller sample sizes. Nonetheless, the observed mean mother-child sequence divergences are unlikely to have been generated through predominantly vertical transmission. It is difficult to distinguish the observed data from either mixed transmission or predominantly horizontal scenarios, given the wide confidence estimates on mean, median and skewness. Skewness, in particular, has confidence limits that encompass both right-skewed ($g_1>0$) and left-skewed distributions ($g_1<0$), which results from the bimodal distribution of mother-child divergences in the observed data (Figure 5).

**Discussion**

Previous population genetic studies of *H. pylori* have shown the existence of ancestral population types or strains that are consistent with geographic regions (30,33). These studies indicate the effects that human migrations have had on global genetic diversity within *H. pylori*. Evolutionarily, the existence of multiple strains, each with an ancestral origin that can be geographically determined, provides an independent marker from that of Y-chromosome, microsatellite or mitochondrial DNA studies, for deciphering human history (34). Comparisons of these markers show that DNA sequences from *H. pylori* provide greater resolution, for example, in the separation of Buddhists and Muslim populations in Ladakh, India, than do mtDNA sequences or microsatellites (34). The accurate inference of human migratory patterns using *H. pylori*, however, has been justified on the assumption of strictly vertical transmission (34). To date most studies have suggested predominantly vertical transmission (14,16,18), with most evidence for infection contained within the family unit and a sampling bias towards maternal transmission. We have outlined the methodological problems of these studies and have shown, using simulation modeling, that in a high-prevalence population, transmission of *H. pylori* is not strictly vertical and includes a strong horizontal component derived from the community. Many people in developing countries, and until recently those in the developed countries, live in comparable social conditions and experience similarly high *H. pylori* prevalence to this community (7, 20, 21). Thus the retention of strong ancestral geographic structure within global *H. pylori* sequences (30), within sequences from regional populations (34), and within the single homogeneous community considered in this study requires an alternate explanation. The ancestral population structure observed in Ladakh, is most likely the result of cultural separation of these religious population groups (35,36). The persistence of ancestral *H. pylori* lineages within the essentially homogeneous and intermarried community in the current study, however, suggests that this structure is maintained by bacterial interactions rather

than through separation of human societies alone.

Phylogenetic structuring of global *H. pylori* genotypes probably arose through the isolation of ancestral human populations before the onset of migration and admixture. Subsequent recombination between lineages will disrupt this ancestral population structure but recombination within lineages tends to retain this ancestral structure by homogenizing within group differences. Most likely this retention of ancestral population structure is the result of genome selective mechanisms, which act to limit recombination between different ancestral population groups. Indeed recent microarray experiments show that isolates maintain a genomic framework or scaffold where genes may be shuffled, lost or gained and recombination occurs intra- and extra-genomically, but seldom inter-genomically (37).

From a medical perspective, the inference of predominantly horizontal transmission, from outside the family, affects our epidemiological understanding of *H. pylori* infection, especially in high prevalence communities. The very high levels of gene diversity and the observation that most individuals carry highly divergent genotypes imply the presence of an immense community reservoir of *H. pylori* genotypes, that serves as a source of infection. Many studies have searched for environmental sources of *H. pylori* such as water supplies (13,38,39), food (40), or insect vectors (41,42), but these sources remain poorly substantiated and controversial, perhaps due to the presence of an alternate form of the bacterium that is difficult to culture (5,43). The Ogies community considered here has a reticulated supply of treated tap-water and flushing toilets, hence water and sanitation are unlikely explanations for the high prevalence and diversity of *H. pylori* in this population. An alternative reservoir may be within the community itself, with infection passed from person to person, especially among children. Existing studies have found little evidence for transmission between school children (18) but these results were based on serological analysis in a low prevalence community. Such transmission pathways could account for the strong community-derived transmission component observed in the current study.

**Acknowledgments**

This work was presented at a plenary session of the Digestive Disease Week conference in May 2004. We would like to thank Mark Achtman, John Atherton and Daniel Falush for pertinent comments on an earlier draft of this manuscript. Wayne Delport would like to thank Prof. P. Bloomer and Prof. J. W. H. Ferguson, for their support. Dr Schalk van der Merwe is a recipient of Astra-Zeneca/ South African Gastroenterology Society Fellowship in Gastroenterology.

**References**

1. Goodman, K. J. & Correa, P. (1995) *Int. J. Epidemiol.* **24**,875-877.
2. Alm, R. A., Bina, J., Andrews, B. M., *et al.* (2000) *Infect. Immun.* **68**, 4155-4168.
3. NIH Consensus Development Panel. (1994) *J. Am. Med. Assoc.* **272**, 65-9.
4. Covacci, A., Telford, J. L., Del Giudice, G., *et al*. (1999) Science **284**, 1328-33.
5. Dowsett, S. A. & Kowolik, M. J. (2003) *Crit. Rev. Oral Biol. Med.* **14**, 226-233.
6. Goosen, C., Theron, J., Ntasala, M., Maree, F. F., Olckers, A., Botha, S. J., Lastovica, A.J., Van der Merwe, S. W. (2002) *J Clin Microbiology* **40**, 205-209.
7. Olivier, B. J., Bond, R. P., Van Zyl, W. P., Delport, M., Slavik, T., Ziady, C., Terhaar, J., Veldboer, C., Lips, M., Lastovica, A., Fransen, J. H., Kusters, J. G. Kuipers, E. J., Van der Merwe, S. W. (in press) Helicobacter
8. Thomas, J. E., Gibson, G. R., Darboe, M. K., Dale, A., Weaver, A. T. (1992) *Lancet* **340**, 1194-1195.
9. Kelly, S. M., Pitcher, M. C. L., Farmery, S. M., Gibson, G. R. (1994) *Gastroenterology* **107**, 1671-1674.
10. Parsonnet, J., Shmuely, H., Haggerty, B. S. (1999) *J. Am. Med. Assoc.* **282**, 2240-2245.
11. Alm, R. A., Ling, L. S., Moir, D. T., *et al.* (1999) *Nature* **397,**176-180.
12. Suerbaum, S., Michetti, P. (2002) *New Eng. J. Med.* **347**, 1175-1186.
13. Bunn, J. E. G., MacKay WG, Thomas JE, *et al.* (2002) *Lett. Appl. Microbiol.* **34**, 450-454.
14. Drumm, B., Perez-Perez, C. I., Blaser, M. J., Sherman, P. M. (1990) *New England Journal of Medicine* **322**, 359-363.
15. Bamford, K. B., Bickley, J., Collins, J. S. A. (1993) *Gut* **34**, 1348-1350.
16. Rothenbacher, D., Bode, G., Berg, G. *et al.* (1999) *J. Infect. Dis.* **179**, 398-402.
17. Samir, K. S., Martin, B., Gold, B. D. *et al.* (2004) *Helicobacter* **9**, 59-68.
18. Tindberg, Y., Bengtsson, C., Granath, F. *et al*. (2001) *Gastroenterol.* **121**, 310-316.
19. Everhart, J. E. (2000) *Gastroenterol. Clin. North Am.* **20**, 559-578.
20. Parsonnet, J., Blaser, M. J., Perez-Perez, G. I. *et al.* (1992) *Gastroenterology* **102**, 41-46.
21. Banatvala, N., Mayo, K., Megraud, F. (1993) *J. Infect. Dis.* **168**, 219-221.
22. Wang, J. T., Sheu, J. C., Lin, J. T. *et al.* (1993) *J. Infect. Dis.* **168,** 1544-1548.
23. Prewett, E. J., Bickley, J., Owen, R. J., Pounder, R. E. (1992) *Gastroenterology* **102**, 829-833.??
24. Pritchard, J. K., Stephens, M., Donnelly, P. (2000) Genetics **155**, 945-959.
25. Miehlke, S., Thomas, R., Guiterrez, O., Graham, D. Y., Go, M. F. (1999) *J. Clin. Microbiol.* **37**, 245-247.
26. Luman, W., Zhao, Y., Ng, H. S., Ling, K. L. (2002) *European Journal of Gastroenterology and*

*Hepatology* **14**, 521-528.

27. Akashi, H., Hayashi, T., Koizuka, H., Shimoyama, T., Tamura, T. (1996) *J. Gastroenterol.* **31** (Suppl ix), 16-23.
28. Rozas, J., Rozas, R. (1999) *Bioinformatics* **15**, 174-175.
29. Hudson, R. R. & Kaplan, N. L. 1985. *Genetics* **111**, 147-164.
30. Falush, D., Wirth, T., Linz, *et al.* (2003) *Science* **299**, 1582-1585.
31. Falush, D., Stephens, M. & Pritchard, J. K. (2003) *Genetics* **164**, 1567-1587.
32. Swofford, D. L. (1999) Phylogenetic Analysis Using Parsimony (PAUP), version 4.0. Illinois Natural History Survey, Champaign.
33. Achtman, M., Azuma, T., Berg, D. E., Ito, Y., Morelli, G., Pan, Z-J., Suerbaum, S., Thompson, S. A., van der Ende, A., van Doorn, L-J. (1999) *Mol. Microbiol*. **32**, 459-470.
34. Wirth, T., Wang, X., Linz, B. *et al.* (2004). *Proc. Natl. Acad. Sci. USA* **101**, 4746-4751.
35. Kaul, S. & Kaul, H. N. (1992) in Ladakh Through the Ages, Towards a New Identity. (Nataraj Books, Springfield, VA), pp 118-141.
36. Srinivas, S. (1998) The Mouths of People, the Voice of God (Oxford Univ. Press, New York).
37. Raymond, J., Thiberge, J., Kalach, N., Bergeret, M., Dauga, C., Labigne, A. (2004) *Helicobacter*, **9**, 492
38. Hultzen, K., Han, S. W., Enroth, *et al.* (1996) *Gastroenterology* **110**, 1031-1035.
39. Klein, P. D., Graham, D. Y., Gaillour, A. *et al.* (1991) *Lancet* **337**, 1503-1506.
40. Hopkins, R. J., Vial, P. A., Ferrecio, C., *et al.* (1993) *J. Infect. Dis.* **168**, 222-226.
41. Grubel, P., Huang, L., Masubuchi, N. *et al.* (1998) *Lancet* **352**, 788-789.
42. Osato, M. S., Ayub, K., Le, H. *et al.* (1998) *J. Clin. Microbiol.* **36**, 2786-2788.
43. Bode, G., Mauch, F., Maltfertheiner, P. (1993) *Epidemiol. Infect.* **111**, 483-490.
44. Rosenberg, N. H. (2004) *Molecular Ecology Notes* **4**, 137-138.

# A population genetics pedigree perspective on the transmission of *Helicobacter pylori*

Wayne Delport[1], Michael Cunningham[1], Brenda Olivier[2], Oliver Preisig[3] & Schalk W van der Merwe[2,4]

[1]Molecular Ecology and Evolution Programme, Department of Genetics, University of Pretoria, Pretoria, 0002, South Africa
[2]Hepatology/GI-research laboratory, Department of Internal Medicine, University of Pretoria, Pretoria, 0002, South Africa
[3]Inqaba Biotech, Pretoria, South Africa

## Supporting Materials and Methods

*Gene sequencing*

These housekeeping genes were chosen, using the calculations of recombination end-point frequency from (1) to span both narrow (UreC-Ure1) and moderate (UreC-MutY) genomic distances. This strategy would facilitate detection of recombination events following transmission between sampled individuals, a process that could potentially obscure patterns of vertical inheritance of *H. pylori* infection. The inter gene distances between UreI and UreC may be sufficient to maintain linkage disequilibrium between these fragments, while the greater distance between UreC and MutY would allow more recombination and, potentially, break down of linkage disequilibrium. PCR products were purified by precipitation with 95% Ethanol and 3M NaAc, and sequence reads were determined on an ABI 3100 capillary sequencer, following cycle-sequencing using the BigDye 3.2 termination reaction.

*Sequence analysis*

Sequences from the three genes, UreI, UreC & MutY with 80, 79 and 79 genotypes respectively, were imported into Sequence Navigator (Applied Biosystems, California), where they were proof-read and subsequently aligned using ClustalX (2). We performed preliminary analyses of sequence diversity in DnaSp (3), and used Mega2 (4) to construct unrooted neighbour-joining phylograms based on uncorrected *p* distances for each of the three genes. The aim of these preliminary analyses was to identify overall trends, and population genetic structure within the dataset derived from the Ogies community. Subsequent model-based analyses aimed (i) to align the population genetic structure of *H. pylori* in this community with the previously observed global structure (5), (ii) to estimate the degree of recombination across the three genes sequenced, and (iii) to test alternate transmission hypotheses using the gene sequence and pedigree data from the Ogies community. We

address these aims using Bayesian clustering assignment methods, Bayesian estimation of sample parameters and a custom-built transmission simulation model, respectively.

*Global H. pylori diversity and population structure*

Falush et al. (5) utilised a Bayesian approach (using Structure 2 (6)) to assign global samples of *H. pylori* to ancestral populations using eight loci. The authors performed these assignments using either a no admixture model, which assumes that each individual has derived its ancestry from a single population, or an admixture model that incorporates recombination between nucleotides. In the former model four modern population groups were consistently identified as hpAfrica1, hpAfrica2, hpEurope and hpEastAsia, yet the hpEurope group showed considerable mixed ancestry, probably the result of Europe being populated by several independent migrations of unknown genetic origin (5,7,8). The incorporation of admixture into the assignment model, which assigns individual nucleotides to ancestral populations based on their linkage with adjacent nucleotides, consistently identified five ancestral population clusters namely, ancestral Africa1 (AA1), ancestral Africa2 (AA2), ancestral Europe1 (AE1), ancestral Europe2 (AE2) and ancestral EastAsia (AEA).

The purpose of our analysis was to determine the contribution of these modern and ancestral population groups to the sample of *H. pylori* sequences from a single rural African community. To this end we first identified individuals from a previous data set (5) that were consistently assigned to their respective extant population groups given the genes that were sequenced in this study (UreI, UreC & MutY). The hpEastAsia, hpAfrica2 and hpAfrica1 groups show little to no recombination among modern populations for two of the genes sequenced in this study, but the hpEurope population group shows high levels of recombination, with the genome of each isolate comprising a mosaic of small recombinant fragments (5: Figure 2). Therefore, we excluded European individuals from the no admixture model for assignment of genotypes to modern global population groups. This exclusion is supported by the relatively low proportion of nucleotides derived from hpEurope in the black South African samples sequenced previously (5). The no admixture model was implemented in Structure 2 (6), with correlated allele frequencies, a burn-in period of 10000 iterations, and subsequent sampling for 20000 iterations. In order to assign sequences to ancestral populations, and to allow for mixed ancestry, we used an admixture model in Structure2 (6) which allows separate assignment of ancestry to each nucleotide variant. These analyses were performed with the full dataset from Falush *et al.* (5), including European individuals, combined with the data generated in this study. The admixture model was implemented with correlated allele frequencies, for five subpopulations (as determined in 5) and a burn-in period of 10000 iterations, with a subsequent 20000 iterations of sampling.

*Within-community population structure*

A second aim of these analyses was to test for the existence of population genetic structure within a single community of *H. pylori* infected individuals, and to estimate the rate of recombination and average size of recombinant chunks based on these data. Structure 2 estimates the number of distinct subpopulations given the observed data. This estimate is based on the premise that linkage disequilibrium should be observed between populations, but rarely within populations (6). In order to assign individuals to *K* subpopulations we performed heuristic analyses with *K* in the range from two to seven, and noted the posterior probabilities of the data given the value of *K*. These posteriors are dubious at best (6) but they can still be used as an *ad hoc* guide to decide which models best fit the data (6). These exploratory simulations were each run with a burn-in of 10000 iterations, and sampling for a further 20000 iterations (Table 4). We then proceeded with the analyses of structure within the Ogies study population under a linkage and admixture model (as in the analysis of global structure), using the number of groups which best fit the data under each of these models.

*Transmission analyses*

To obtain a preliminary view of the information content of the data in terms of analyzing transmission hypotheses across relationship categories we performed Chi-square permutation tests. These tests examined the association between individuals carrying similar sequences and individuals within a particular category of relationship (mother-child, parent-offspring, siblings, family members or housemates). A permutation approach was necessary to assess the significance of the observed Chi-square test statistic as each individual appears in multiple pairwise comparisons, thus these comparisons are not independent of each other. We used a computer program incorporating the Roff & Bentzen algorithm (9) to produce a series of 1000 randomized test matrices for each comparison, with the number of similar sequences and the number of individuals in the particular relationship category (the marginal values) held constant. To assess the significance of an association we determined the proportion of these randomized matrices that showed more extreme Chi-square values than the observed data. Since mutations may have accumulated since the time of transmission, we scored sequences as similar if they differed by five or fewer substitutions. This criterion was based on a gap observed in the distribution of pairwise sequence differences (the mismatch distribution), with few comparisons that differ by between six to ten nucleotide substitutions. The maximum mutation rate of *H. pylori* has been estimated as $2.28e^{-5}$ mutations per site per year (5). Given a total of 1391 sites sequenced in this study, the probability of a single mutation occurring each year is 0.03. Thus the time required for five mutations to occur is on the order of 160 years, and it is highly unlikely that more than five mutations could have accumulated between the times of transmission and sampling.

Categorical tests alone provide an inadequate representation of *H. pylori* transmission patterns as there may be multiple routes of person to person infection. A probabilistic model of infection, with pedigree and associated sequence data as inputs, was constructed to characterize patterns of genetic diversity expected under more complex transmission scenarios. In the model, transmission was simulated using a broken-stick design (Figure 2), where the source of infection for each target individual was determined by a random draw from a uniform distribution, with varying frequency segments assigned to five relationship categories (Table 1). Given the mode of transmission, the source individual and the associated DNA sequence was identified from the pedigree data. Infection from the community was simulated by drawing from the dataset of available individuals (n=74), and creating a new allele at a rate determined by the observed gene diversity, that is, from the expectation that two randomly chosen sequences are different. The number of nucleotide substitutions to be enforced along this new allele was determined by first randomly choosing one of the three identified phylogenetic groups (Figure 3) at a rate determined by their frequency in the observed data, and then choosing a pairwise difference from the mismatch distribution within these groups for each of the three candidate genes. Substitutions along the new allele were applied according to the appropriate mutation model for each gene as determined in PAUP*4b10 (10) using ModelTest 3.4 (11) and AIC model selection. Substitutions were constrained to the observed variable sites, or to new mutant sites arising at a rate determined by the nucleotide diversity, such that the strong phylogenetic structure evident in *H. pylori* (5) would be retained in the model. Age of infection was drawn from a gamma distribution (mean = 3, alpha = 0.1), and used to calculate the sequence divergence from time of infection to time of sampling. Time since divergence determined the number of mutation and recombination events, with rates of $6.9e^{-5}$ and $4.1e^{-5}$ per nucleotide per annum respectively (1). The probability of mutation and recombination events along the total sequence length of 79440bp spanning the three genes considered in this study was calculated as $1-(1-\alpha)^{79440}$, where $\alpha$ represents the per site per annum mutation or recombination rate. The occurrence of recombination or mutation events was determined by a random draw for both the probability of occurrence and the affected nucleotide along the entire sequence length. Mutation events that occurred within the sequenced gene regions (Ure1, UreC & MutY) were enforced according to the best-fit mutation model for each gene. Similarly, recombination events that occurred within the sequenced gene regions, or that occurred within one recombinant's length of any of these gene regions, drawn from an exponential distribution with a mean of 417bp (1), were enforced along the simulated DNA sequences. Transmission was simulated for 26 individuals, for whom at least one parent's bacterium infection had been sequenced, and was repeated 1000 times for each of seven transmission hypotheses (Table 1). The model only simulated a single transmission event per individual, and thus summary statistics (gene diversity and nucleotide diversity) and phylogenetic structure of the simulated data should not differ substantially under the

alternate transmission hypotheses. However, pairwise-comparisons of parent-child divergence should show marked differences. We performed Archie-Faith-Cranston randomisation tests, with 1000 permutations, in PAUP*4b10 (9), which calculated the probability that the topologies of randomly chosen simulated datasets for each transmission scenario were consistent with that of the observed data topology. To compare results from alternate transmission scenarios we calculated the mean, median and skewness from simulated distributions of mother-child divergences, and used jackknife, with 50% deletion, and bootstrap resampling procedures to calculate confidence limits on these statistics for each of the seven transmission hypotheses. Finally, we used these resampling procedures along with Kolmogorov-Smirnov tests to infer whether any of the seven transmission hypotheses were unlikely to have given rise to the observed data.

**References**

1. Falush, D., Kraft, C., Taylor, N. S., *et al.* (2001) *Proc. Natl. Acad. Sci. USA* **98**, 15056-16061.
2. Higgins, D. G., Sharp, P. M. (1988) *Gene* **73**, 237-244.
3. Rozas, J., Rozas, R. (1999) *Bioinformatics* **15**, 174-175.
4. Kumar, S., Tamura, K., Nei, M. (2004) *Briefings in Bioinformatics* **5**, 150-163.
5. Falush, D., Wirth, T., Linz, *et al.* (2003) *Science* **299**, 1582-1585.
6. Pritchard, J. K., Stephens, M., Donnelly, P. (2000) *Genetics* **155**, 945-959.
7. Semino, O., Passarino, G., Oefner, P. J., et al. (2000) *Science* **290**, 1155-1159.
8. Chikhi, L., Nichols, R. A., Barbujani, G. & Beaumont, M. A. (2002) *Proc. Natl. Acad. Sci. USA* **99**, 11008.
9. Roff, D. A., Bentzen, P. (1989) *Mol. Biol. Evol.* **6**, 539-545.
10. Swofford, D. L. (1999) Phylogenetic Analysis Using Parsimony (PAUP), version 4.0. Illinois Natural History Survey, Champaign.
11. Posada, D., Crandall, K. A. (1998) *Bioinformatics* **14**, 817-818.

**Table 4**: Inference of the number of subpopulations within the Ogies community sample. ln P (X|K) is the log likelihood of the data given the specified K. Log likelihoods are given for five separate simulations for each value of K.

| **K** | **ln *P*(X\|K)** | | | | | **mean ln *P*(X\|K)** |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | |
| 2 | -4609 | -4609 | -4613 | -4609 | -4609 | -4610 |
| 3 | -3923 | -4231 | -3786 | -3780 | -3786 | -3901 |
| 4 | -3527 | -3570 | -3663 | -3498 | -3599 | -3571 |
| 5 | -3839 | -3397 | -3255 | -3448 | -3410 | **-3470** |
| 6 | -3215 | -3042 | -5465 | -3045 | -7091 | -4372 |
| 7 | -3473 | -3159 | -4233 | -3402 | -4535 | -3760 |

**Table 5:** Bootstrap estimates and 95% confidence limits (lower, upper) for distribution statistics from simulated and observed data. Y = mean, M = median, $g_1$ = skewness

a) Mother-child sequence divergences from simulated data under the seven transmission models.

| a) | ***Y*** | ***M*** | $g_1$ |
|---|---|---|---|
| **1** | 0.0281(0.0276, 0.0285) | 0.0134 (0.0125, 0.0143) | 0.7135 (0.6866, 0.7135) |
| **2** | 0.0315 (0.0311, 0.0320) | 0.0183 (0.0172, 0.0193) | 0.5238 (0.4984, 0.5492) |
| **3** | 0.0374 (0.0370, 0.0379) | 0.0305 (0.0293, 0.0317) | 0.2601 (0.2356, 0.2846) |
| **4** | 0.0418 (0.0413, 0.0423) | 0.0403 (0.0389, 0.0417) | 0.0590 (0.0357, 0.0823) |
| **5** | 0.0361 (0.0356, 0.0366) | 0.0251 (0.0240, 0.0261) | 0.3230 (0.2981, 0.3478) |
| **6** | 0.0411 (0.0407, 0.0416) | 0.0383 (0.0364, 0.0401) | 0.1405 (0.1165, 0.1645) |
| **7** | 0.0527 (0.0523, 0.0531) | 0.0563 (0.0544, 0.0571) | -0.3282 (-0.3503, -0.3061) |

b) Distribution statistics of mother-child (n = 19), father-child (n = 16), sibling (n = 34) and within-household (n = 99) sequence divergences from observed data.

| b) | ***Y*** | ***M*** | $g_1$ |
|---|---|---|---|
| **Mother-child** | 0.0463 (0.0335, 0.0591) | 0.0493 (0.0243, 0.0743) | -0.0478 (-0.8343, 0.7385) |
| **Father-child** | 0.0489 (0.0397, 0.0582) | 0.0444 (0.0291, 0.0598) | 0.7750 (-0.2901, 1.8402) |
| **Sibling** | 0.0309 (0.0217, 0.0401) | 0.0273 (0.0054, 0.0492) | 0.2819 (-0.2793, 0.8430) |
| **Household** | 0.0406 (0.0364, 0.0449) | 0.0418 (0.0345, 0.0491) | 0.0879 (-0.2161, 0.3920) |

**Table 6:** Kolmogorov-Smirnov tests comparing observed versus simulated distributions of test statistics under seven transmission scenarios (Table 1). k = Kolmogorov-Smirnov test statistic, p = probability that the observed distribution could be generated by the simulated model. Significance at the 0.05 level is indicated in bold.

| | **Mother-child** | | **Father-child** | | **Sibling** | | **Household** | |
|---|---|---|---|---|---|---|---|---|
| | k | p | k | p | k | p | k | p |
| 1 | 0.395 | **0.004** | 0.447 | **0.002** | 0.150 | 0.643 | 0.828 | **<<0.001** |
| 2 | 0.346 | **0.016** | 0.537 | **<0.000** | 0.113 | 0.915 | 0.811 | **<<0.001** |
| 3 | 0.271 | 0.103 | 0.294 | 0.102 | 0.180 | 0.407 | 0.891 | **<<0.001** |
| 4 | 0.216 | 0.303 | 0.379 | **0.014** | 0.164 | 0.526 | 0.888 | **<<0.001** |
| 5 | 0.288 | 0.070 | 0.436 | **0.003** | 0.161 | 0.553 | 0.790 | **<<0.001** |
| 6 | 0.183 | 0.508 | 0.337 | **0.040** | 0.273 | 0.053 | 0.748 | **<<0.001** |
| 7 | 0.153 | 0.729 | 0.244 | 0.259 | 0.185 | 0.377 | 0.925 | **<<0.001** |

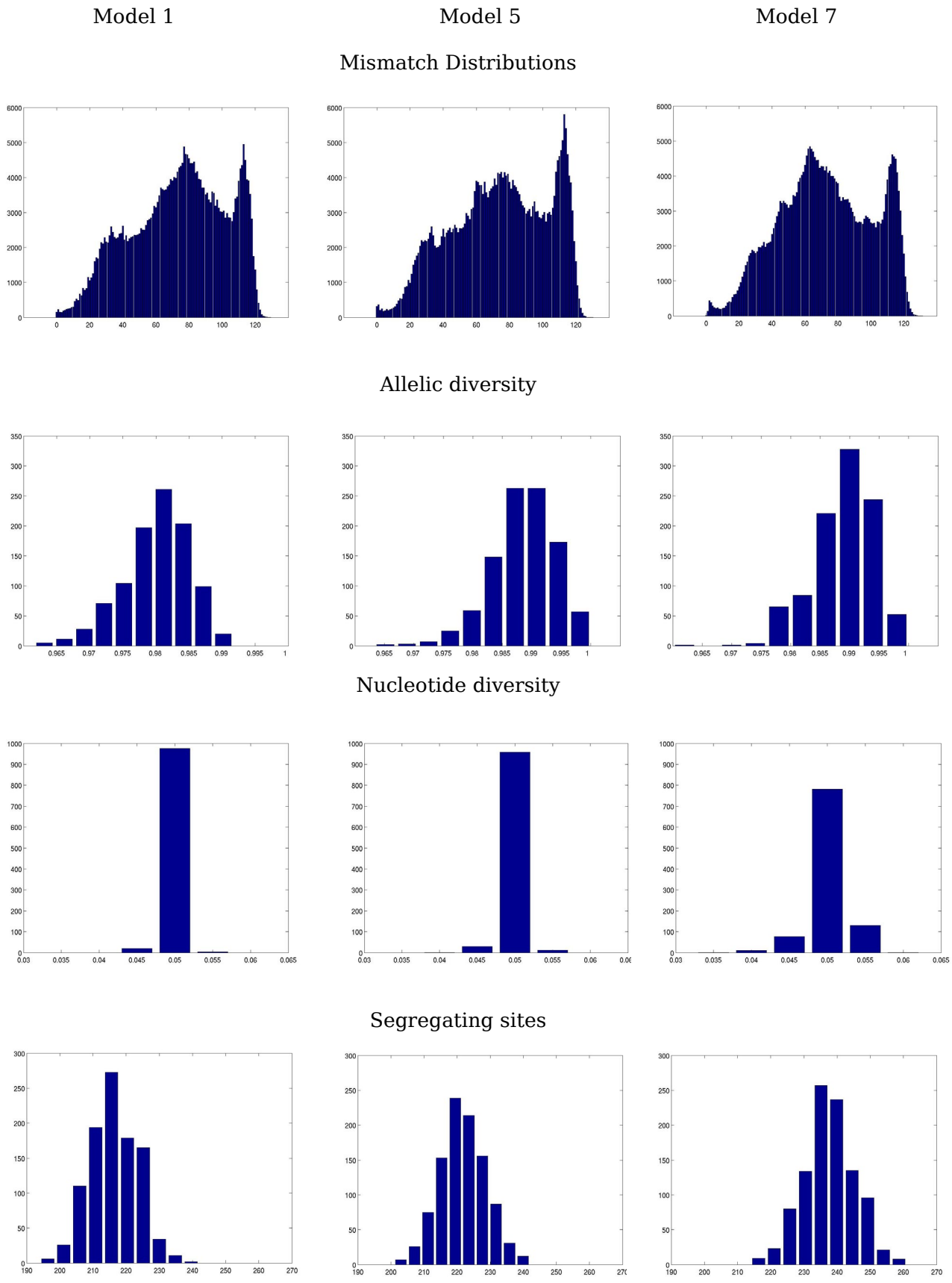


**Figure 6:** Mismatch distributions, allelic and nucleotide diversity distributions, and distributions of segregating sites under three alternate transmission hypotheses. These results show that the underlying phylogenetic structure of *H. pylori* evident within the sampled community is comparable to that in the simulations.

# Appendix II

## LatticeFlucIII C Source code

"640K ought to be enough for anybody"

Bill Gates, 1981

```
/*LatticeFlucIII program simulates population turnover in a
continously distributed species for a given number of microsatellite
loci. The program can either be run with unstable of stable
population size, where population size at each generation in the fo
rmer is determined by a Ricker logistic growth model, with a random
draw for growth rate. Rousset's genetic distance between individuals
is calculated at each generation for x subpopulations, each with
population size y. These distances are regressed agai
snt the log of distance to provide an estimate of neighbourhood size.
The purpose of this program is to observe the effect of population
size fluctuations on the estimation of neighbourhood size, using
Rousset's methods.

LatticeFlucIII differs from LatticeFlucII in that it allows the input
of a population size array for population size fluctuations. In this
way the same stochastic simulations can be repeated n times to test
for variance in the observed statistics. Also choice is given for the
dispersal distribution to be leptokurtic or normal.

Rousset (1997) Genetic differentiation and estimation of gene flow
from F-statistics under isolation by distance. Genetics 145: 1219-
1228.
Rousset (2000) Genetic differentiation between individuals. J. Evol.
Biol. 13: 58-62.

Wayne Delport
Molecular Ecology and Evolution Programme
Department of Genetics
University of Pretoria
Pretoria
0002
South Africa
wdelport@postino.up.ac.za

May 2005
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define PI 3.141592654

int SetStartingConditions ( int, int, int, int, float, float );
int AssignSex ( int, int, int );
int PopSizeCalc ( int, float, float, int, int );

float gammaMultiplication ( float );
float gammaRejection ( float );
float gammaTwoPart ( float );
float normal ( float );
float sumofsquares ( float[], float, int );
float mean ( float[], int);

void InitialiseArrays ( int, int, int, int, int, int );
void CreateTempArrays ( int, int, int );
void AssignMuRate ( float, float, int );
void ResetTempArrays ( int, int, int );
void Reproduction ( int, int, int, int, int, int );
void ReproductionII ( int, int, int, int, float, int, int );
void ReproductionIII ( int, int, int, int, float, int, float, int,
```

```
int );
void SamplePopulation ( int, int, int, int, int, int );
void SamplePopulationII ( int, int, int, int );
void WriteDataMatrix ( int, int, int );
void LogOfDistance( float[], int );
void WriteOutfile( int, char[], int );
void PreRegression ( int, int );
void Regression ( float[], float[], float[], int );
void CountAlleles ( int, int );
void CalculateAr ( int, int );
void WritePopMatrix ( int, int, int, int );
void CalculateDiversity ( int, int );
void CalculateAllelicDiversity ( int, int );
void CalculateAlleleDistribution ( int, int, int, int );
void ExportData ( char[], int, int, int );
void ExportSharedAlleleDistances ( char[], int, int, int );
void InitialiseSampleArrays ( int, int );

FILE *ptrOutfile;
FILE *ptrOutfile2;
FILE *ptrInfile;
FILE *ptrStartFile;
FILE *ptrFlucFile;
FILE *ptrDispersalFile;
FILE *ptrOutfile3;

int ****Data;
int ****dataTemp;
int **sexData;
int **sexTemp;
int **Data1, **Data2;
int **Alleles;
int *DataRow, *DataCol;
int *NodeRows;
int *NodeCols;
int *SampledRows;
int *SampledColumns;
int *LocusSSW;
int *LocAlleleNumbers;
int *PopSizeArray;
int *PopFlucArray;
int NodesOccupied;
int reps;
int ActualN;
int NewN;
int popsize;
int WriteDispersalFile;
int WriteSharedAlleleDistances;

float *MutationRate;
float *LogDistance;
float *AdiArray;
float *AdiSDArray;
float *NewLogDistance;
float *ArArray;
float *NewArArray;
float *ArWithinYear;
float *AhatArray;
float *AhatWithinYear;
float *NewAhatArray;
float *DistanceArray;
```

```
float *HeWithinYear;
float *AdiWithinYear;
float *AllelicDivWithinYear;
float *TotalDiv;
float *TotalDivSD;
float *NbSizeAr;
float *NbSizeArSD;
float *NbSizeAhat;
float *NbSizeAhatSD;
float *TotalAllelicDiv;
float *TotalAllelicDivSD;
float *MeanDispersal;
float ***SharedDistances;
float percentOccupied;
float NbSizey1, NbSizey2;
float H;
float AllelicDiv;
float dVariance;
float dVariance2;
float Adi;

char NameString [ 100 ];

int main (int argc, char *argv[ ])
{
  int LatticeRows, LatticeCols;
  int NumLoci;
  int MeanNumAllelesPerLoc;
  int dParameter;
  int dParameter2;
  int RandomRow1, RandomCol1;
  int RandomRow2, RandomCol2;
  int occupied;
  int male;
  int nyears;
  int empty, counter;
  int RandomOffspringX, RandomOffspringY;
  int SampleSize;
  int pCounter;
  int i,j,k,l,m,n,o,p;
  int NodesOccupiedPrev;
  int WriteSampleFiles;
  int DemographicStability;
  int StartingPopulation;
  int Allele1, Allele2, Sex;
  int PopsToSample;
  int PopExtent;
  int PopCrash;
  int DispersalDistrib;
  int CalcMethod;


  float alleleAlpha;
  float MuRateAlpha, MuRateBeta;
  float DirectionY, DirectionX;
  float r;
  float VarEt;
  float meanAr, meanAhat, meanHe, meanAllelicDiv, meanAdi;
  float sdAr, sdAhat, sdHe, sdAllelicDiv, sdAdi;
  float dist, dist2, ar, ahat;
  float testfloat;
```

```
  float Heterozygosity;

  char* DataExport = NULL;
  char* SharedAlleleDistances = NULL;

  if ( argc!=8 )
    printf( "Usage: LatticeFlucIII infile outfile1 outfile2
PopStartFile PopFlucFile DispersalFile SharedAllelesFile\n" );
  else {
    if ( ( ptrInfile = fopen ( argv [ 1 ], "r" ) ) == NULL )
      printf( "Could not open parameter file\n" );
    else {
      if ( ( ptrOutfile = fopen ( argv [ 2 ], "a" ) ) == NULL )
    printf( "Could not open outfile\n" );
      else {
    srand( time( NULL ) );
    DataExport = argv[ 3 ];
    SharedAlleleDistances = argv [ 7 ];
    fscanf ( ptrInfile, "%d%d\n", &LatticeRows, &LatticeCols );
    fscanf ( ptrInfile, "%d\n", &StartingPopulation );
    fscanf ( ptrInfile, "%d\n", &DemographicStability );
    fscanf ( ptrInfile, "%f\n", &percentOccupied );
    fscanf ( ptrInfile, "%d\n", &nyears );
    fscanf ( ptrInfile, "%d\n", &DispersalDistrib );
    if ( DispersalDistrib == 0 ) {
      fscanf ( ptrInfile, "%d\n", &dParameter );
      //printf ( "DispersalDistrib = 0\n" );
    }
    if (DispersalDistrib == 1 ) {
      fscanf ( ptrInfile, "%d%f\n", &dParameter, &dVariance );
      //printf ( "DispersalDistrib = 1\n" );
    }
    if ( DispersalDistrib == 2 ) {
      fscanf ( ptrInfile, "%d%f%d%f\n", &dParameter, &dVariance,
&dParameter2, &dVariance2 );
      //printf ( "DispersalDistrib = 2\n" );
    }
    fscanf ( ptrInfile, "%f\n", &r );
    fscanf ( ptrInfile, "%f\n", &VarEt );
    fscanf ( ptrInfile, "%d\n", &NumLoci );
    fscanf ( ptrInfile, "%f\n", &Heterozygosity );
    fscanf ( ptrInfile, "%d%f", &MeanNumAllelesPerLoc, &alleleAlpha
);
    fscanf ( ptrInfile, "%f%f\n", &MuRateAlpha, &MuRateBeta );
    fscanf ( ptrInfile, "%s\n", NameString );
    fscanf ( ptrInfile, "%d\n", &CalcMethod );
    fscanf ( ptrInfile, "%d\n", &SampleSize );
    fscanf ( ptrInfile, "%d\n", &PopsToSample );
    fscanf ( ptrInfile, "%d\n", &PopExtent );
    fscanf ( ptrInfile, "%d\n", &WriteSampleFiles );
    fscanf ( ptrInfile, "%d\n", &WriteDispersalFile );
    fscanf ( ptrInfile, "%d\n", &WriteSharedAlleleDistances );
    printf ( "Lattice Parameters\n" );
    printf ( "------------------\n" );
    printf ( "LatRows = %d, LatCols = %d\n\n", LatticeRows,
LatticeCols );
    printf ( "Microsatellite parameters\n" );
    printf ( "-------------------------\n" );
    printf ( "Number of Loci = %d\n", NumLoci );
    printf ( "Mean heterozygosity = %f\n", Heterozygosity );
    printf ( "Mean # alleles per locus = %d\n", MeanNumAllelesPerLoc
```

```
);
    printf ( "Allelic diversity alpha parameter = %f\n", alleleAlpha
);
    printf ( "Microsatellite SMM model beta parameters: %f, %f\n\n",
MuRateAlpha, MuRateBeta );
    printf ( "Demographic parameters\n" );
    printf ( "----------------------\n" );
    if ( StartingPopulation == 0 ) {
      printf ( "Started with random population\n" );
      printf ( "Population size is %.0f\n",
LatticeRows*LatticeCols*percentOccupied );
    }
    else
      printf ( "Started with population defined in %s\n", argv [ 4 ]
);
    if ( DemographicStability == 0 )
      printf ( "Constant population size\n" );
    else if ( DemographicStability == 1) {
      printf ( "Population follows Ricker growth model with random
fluctuations in r\n" );
      printf ( "Population growth rate = %f\n", r );
      printf ( "Variance in growth rate = %f\n\n", VarEt );
    }
    else if ( DemographicStability == 2 ) {
      printf ( "Population fluctuations defined in file %s\n", argv [
5 ] );
      if (( ptrFlucFile = fopen ( argv [ 5 ], "r" ) ) == NULL )
        printf( "Could not open FlucFile\n" );
      else {
        PopFlucArray = ( int * ) calloc ( nyears, sizeof( int ) );
        if ( PopFlucArray == NULL )
          printf ( "PopFlucArray memory allocation failure\n" );
        for ( l = 0; l < nyears; l++ ) {
          fscanf( ptrFlucFile, "%d\n", &popsize );
          PopFlucArray [ l ] = popsize;
        }
        fclose( ptrFlucFile );
      }
    }
    printf ( "Total generations = %d\n", nyears );
    if ( DispersalDistrib == 0 ) {
      printf ( "Dispersal drawn from uniform distribution with max =
%d\n", dParameter );
    }
    else if ( DispersalDistrib == 1 ) {
      printf ( "Dispersal drawn from normal distribution with mean =
%d, variance = %f\n", dParameter, dVariance );
    }
    else if ( DispersalDistrib == 2 ) {
      printf ( "Dispersal drawn from leptokurtic distribution with
mean1 = %d, variance1 = %f, mean2 = %d, variance2 = %f\n",
dParameter, dVariance, dParameter2, dVariance2 );
    }
    printf ( "\n" );
    //printf ( "Dispersal = %d\n\n", dParameter );
    printf ( "Calculation of Statistics\n" );
    printf ( "-------------------------\n" );
    if ( CalcMethod == 0 ) {
      printf ( "Sample %d individuals from %d populations each of
lattice size: %d x %d\n", SampleSize, PopsToSample,
PopExtent*dParameter, PopExtent*dParameter );
```

```
    }
    else if ( CalcMethod == 1 ) {
      printf ( "Sample %d individuals from entire lattice for
calculation of statistics, %d repeats for calculation of variance\n",
SampleSize, PopsToSample );
    }
    printf ( "write data file every %d generations\n",
WriteSampleFiles );
    if ( WriteDispersalFile == 1 ) {
      printf ( "Dispersal distances will be written to file %s\n",
argv[ 6 ] );
      printf ( "WARNING: FILES CAN GET VERY LARGE\n" );
    }
    if ( WriteSharedAlleleDistances == 1 ) {
      printf ( "Shared Allele distances will be written to file
%s\n", argv[ 7 ] );
      printf ( "WARNING: FILES CAN GET VERY LARGE\n" );
    }
    InitialiseArrays ( LatticeRows, LatticeCols, SampleSize, NumLoci,
nyears, PopsToSample );
    printf ( "Arrays initialised\n" );
    if ( StartingPopulation == 0 ) {
      SetStartingConditions( NumLoci, LatticeRows, LatticeCols,
MeanNumAllelesPerLoc, alleleAlpha, Heterozygosity );
      AssignSex ( NodesOccupied, LatticeRows, LatticeCols );
    }
    else if ( StartingPopulation == 1 ) {
      if (( ptrStartFile = fopen ( argv [ 4 ], "r" ) ) == NULL )
        printf( "Could not open PopStart File\n" );
      else {
        for ( l = 0; l < LatticeRows; l++ ) {
          for ( m = 0; m < LatticeCols; m++ ) {
        for ( n = 0; n < NumLoci; n++ ) {
          fscanf( ptrStartFile, "%d%d%d\n", &Allele1, &Allele2, &Sex
);
          Data[ l ][ m ][ n ][ 0 ] = Allele1;
          Data[ l ][ m ][ n ][ 1 ] = Allele2;
          //printf( "%d\t%d\t%d\n", Allele1, Allele2, Sex );
          if ( n == 0 )
            sexData[ l ][ m ] = Sex;
        }
          }
        }
        fclose ( ptrStartFile );
      }
    }
    CreateTempArrays ( LatticeRows, LatticeCols, NumLoci );
    AssignMuRate ( MuRateAlpha, MuRateBeta, NumLoci );
    pCounter = ( SampleSize*(SampleSize-1) )/2;
    NodesOccupiedPrev = percentOccupied*(LatticeRows*LatticeCols);
    NodesOccupied = percentOccupied*(LatticeRows*LatticeCols);

    if (( ptrDispersalFile = fopen ( argv[ 6 ], "a" ) ) == NULL )
      printf( "Could not write to dispersal file\n" );
    else {
      PopCrash = 0;
      i = 0;
      while ( ( PopCrash == 0) && ( i < nyears ) ) {
        printf ( "year %d\n", i+1 );
        InitialiseSampleArrays ( PopsToSample, NumLoci );
        ResetTempArrays ( LatticeRows, LatticeCols, NumLoci );
```

```
        if ( DemographicStability == 1 ) { /*1 is for size
fluctuations, 0 for stable population*/
          if ( NodesOccupiedPrev > (LatticeRows*LatticeCols) ) {
        NodesOccupiedPrev = (
percentOccupied*(LatticeRows*LatticeCols) );
          }
          //printf ( "NodesOccupiedPrev is %d\n", NodesOccupiedPrev
);
          NodesOccupied = PopSizeCalc ( NodesOccupiedPrev, r, VarEt,
LatticeRows, LatticeCols );
          //printf ( "Nodes occupied is %d\n", NodesOccupied );
        }
        else if ( DemographicStability == 2 ) {
          NodesOccupied = PopFlucArray [ i ];
          //printf ( "PopFluc from files size is %d\n", NodesOccupied
);
        }
        PopSizeArray [ i ] = NodesOccupied;
        if ( DispersalDistrib == 0 )
          Reproduction ( NodesOccupied, LatticeRows, LatticeCols,
dParameter, NumLoci, i );
        else if ( DispersalDistrib == 1 )
          ReproductionII ( NodesOccupied, LatticeRows, LatticeCols,
dParameter, dVariance, NumLoci, i );
        else if ( DispersalDistrib == 2 )
          ReproductionIII (NodesOccupied, LatticeRows, LatticeCols,
dParameter, dVariance, dParameter2, dVariance2, NumLoci, i );
        WriteDataMatrix ( LatticeRows, LatticeCols, NumLoci );
        for ( o = 0; o < PopsToSample; o++ ) {
          //printf( "PopsToSample is %d\n", PopsToSample );
          if ( CalcMethod == 1 ) {
        //printf ( "SamplePopulationII running\n" );
        SamplePopulationII ( SampleSize, LatticeRows, LatticeCols,
NumLoci ); //entire lattice is sampled
          }
          else if ( CalcMethod == 0 ) {
        SamplePopulation ( SampleSize, LatticeRows, LatticeCols,
NumLoci, PopExtent, dParameter ); //limited geographical extent
          }
          CountAlleles ( ActualN, NumLoci );
          CalculateAllelicDiversity ( ActualN, NumLoci );
          AllelicDivWithinYear [ o ] = AllelicDiv;
          CalculateAr ( ActualN, NumLoci );
          LogOfDistance ( DistanceArray, (ActualN*(ActualN-1))/2 );
          PreRegression ( (ActualN*(ActualN-1))/2, dParameter );
          Regression ( NewLogDistance, NewArArray, NewAhatArray, NewN
);
          ArWithinYear [ o ] = NbSizey1;
          AhatWithinYear [ o ] = NbSizey2;
          CalculateDiversity ( ActualN, NumLoci );
          HeWithinYear [ o ] = H;
          CalculateAlleleDistribution ( ActualN, NumLoci, o, i );
          AdiWithinYear [ o ] = Adi;
          free ( NewLogDistance );
          free ( LogDistance );
          free ( NewArArray );
          free ( NewAhatArray );
          free ( SampledRows );
          free ( SampledColumns );
          free ( DataRow );
          free ( DataCol );
```

```
          free ( Data1 );
          free ( Data2 );
          free ( DistanceArray );
          free ( ArArray );
          free ( AhatArray );
          free ( LocAlleleNumbers );
          free ( Alleles );
          free ( LocusSSW );
        }
        meanAr = mean ( ArWithinYear, PopsToSample );
        meanAhat = mean ( AhatWithinYear, PopsToSample );
        meanHe = mean ( HeWithinYear, PopsToSample );
        meanAllelicDiv = mean ( AllelicDivWithinYear, PopsToSample );
        //printf ( "Allelic div within year = %f\n", meanAllelicDiv
);
        sdAr = sqrt( ( sumofsquares ( ArWithinYear, meanAr,
PopsToSample ) )/(PopsToSample - 1) );
        sdAhat = sqrt( ( sumofsquares ( AhatWithinYear, meanAhat,
PopsToSample ) )/(PopsToSample - 1) );
        sdHe = sqrt( ( sumofsquares ( HeWithinYear, meanHe,
PopsToSample ) )/(PopsToSample - 1) );
        sdAllelicDiv = sqrt ( ( sumofsquares ( AllelicDivWithinYear,
meanAllelicDiv, PopsToSample ) )/(PopsToSample-1) );
        meanAdi = mean ( AdiWithinYear, PopsToSample );
        sdAdi = sqrt ( ( sumofsquares ( AdiWithinYear, meanAdi,
PopsToSample ) )/(PopsToSample-1) );
        NbSizeAr [ i ] = meanAr;
        NbSizeArSD [ i ] = sdAr;
        NbSizeAhat [ i ] = meanAhat;
        NbSizeAhatSD [ i ] = sdAhat;
        TotalDiv [ i ] = meanHe;
        TotalDivSD [ i ] = sdHe;
        TotalAllelicDiv [ i ] = meanAllelicDiv;
        TotalAllelicDivSD [ i ] = sdAllelicDiv;
        AdiArray [ i ] = meanAdi;
        AdiSDArray [ i ] = sdAdi;
        NodesOccupiedPrev = NodesOccupied;
        if ( ( ( (i+1) % WriteSampleFiles ) == 0 ) || ( i == nyears-
1) ) {
          WritePopMatrix ( LatticeRows, LatticeCols, NumLoci, i );
        }
        free ( ArWithinYear );
        free ( AhatWithinYear );
        free ( HeWithinYear );
        free ( AllelicDivWithinYear );
        free ( AdiWithinYear );
        if ( NodesOccupied < 10 ) {
          printf ( "Population crash at generation %d\n", i+1 );
          PopCrash = 1;
        }
        i+=1;
      }
      ExportData ( DataExport, nyears, LatticeRows, LatticeCols );
      if ( WriteSharedAlleleDistances == 1 ) {
        ExportSharedAlleleDistances ( SharedAlleleDistances, nyears,
PopsToSample, SampleSize );
      }
      fclose ( ptrInfile );
      fclose ( ptrOutfile );
      fclose ( ptrOutfile2 );
      if ( WriteSharedAlleleDistances == 1 )
```

```
        fclose ( ptrOutfile3 );
      free ( NodeRows );
      free ( NodeCols );
      free ( TotalDiv );
      free ( TotalDivSD );
      free ( TotalAllelicDiv );
      free ( TotalAllelicDivSD );
      free ( PopSizeArray );
      free ( NbSizeAr );
      free ( NbSizeArSD );
      free ( NbSizeAhat );
      free ( NbSizeAhatSD );
      free ( AdiArray );
      free ( AdiSDArray );
      free ( MeanDispersal );
      if ( WriteSharedAlleleDistances == 1)
        free ( SharedDistances );
      fclose ( ptrDispersalFile );
    }
      }
    }
  }
}

void PreRegression ( int n, int d )
/*According to Rousset 1997 pairwise comparisons of individuals at a
distance less than sigma are not expected to show a linear
relationship. Rousset (1997, 2000) suggests the exclusion of these
comparisons in the calculation of regression.*/
{
  int i;
  int counter;
  int D;

  float lg;
  float ahat, ar;

  NewArArray = ( float * ) calloc ( n, sizeof ( float ) );
  if ( NewArArray == NULL )
    printf ( "NewArArray memory failure\n" );

  NewAhatArray = ( float * ) calloc ( n, sizeof ( float ) );
  if ( NewAhatArray == NULL )
    printf ( "NewAhatArray memory failure\n" );

  NewLogDistance = ( float * ) calloc ( n, sizeof ( float ) );
  if ( NewLogDistance == NULL )
    printf ( "NewLogDistance memory failure\n" );

  counter = 0;
  for ( i = 0; i < n; i++ ) {
    D = DistanceArray [ i ];
    if ( D > d) {
      lg = LogDistance [ counter ];
      NewLogDistance [ counter ] = lg;
      ahat = AhatArray [ counter ];
      NewAhatArray [ counter ] = ahat;
      ar = ArArray [ counter ];
      NewArArray [ counter ] = ar;
      counter+=1;
    }
```

```
  }
  NewN = counter+1;
}

void InitialiseSampleArrays ( int n, int L )
{

  ArWithinYear = ( float * ) calloc ( n, sizeof ( float ) );
  if ( ArWithinYear == NULL )
    printf ( "ArWithinYear memory allocation failure\n" );

  AhatWithinYear = ( float * ) calloc ( n, sizeof ( float ) );
  if ( AhatWithinYear == NULL )
    printf ( "AhatWithinYear memory allocation failure\n" );

  HeWithinYear =  (float * ) calloc ( n, sizeof ( float * ) );
  if ( HeWithinYear == NULL )
    printf ( "HeWithinYear memory allocation failure\n" );

  AllelicDivWithinYear = ( float * ) calloc (n, sizeof ( float ) );
  if ( AllelicDivWithinYear == NULL )
    printf ( "AllelicDivWithinYear memory allocation failure\n" );

  AdiWithinYear = ( float * ) calloc (n, sizeof ( float ) );
  if ( AdiWithinYear == NULL )
    printf ( "AdiWithinYear memory allocation failure\n" );

}

void ExportSharedAlleleDistances ( char a[], int Y, int Pops, int N )
{
  int i,j,k;
  float floatVal;

  if ( ( ptrOutfile3 = fopen ( a, "w" ) ) == NULL )
    printf ( "Could not export Shared Allele Distances\n" );
  else {
    for ( i = 0; i < Y; i++ ) {
      for ( k = 0; k < Pops; k++ ) {
    j = 0;
    floatVal = SharedDistances [ j ][ i ][ k ];
    while ( floatVal != 0 ) {
      if ( j == 0 ) {
        fprintf ( ptrOutfile3, "Gen%dPop%d", i, k );
      }
      fprintf ( ptrOutfile3, ", %f", floatVal );
      j+=1;
      floatVal = SharedDistances [ j ][ i ][ k ];
    }
    fprintf ( ptrOutfile3, "\n" );
      }
    }
  }
}

void ExportData ( char a[], int Y, int Rows, int Cols )
{
  int i;
  int pSize;
  float Tdiv, TdivSD, Ldiv, LdivSD, Nb1, Nb2, D, d;
  float Nb1SD, Nb2SD, ADiv, ADivSD, ADI, ADISD;
```

```
  if ( ( ptrOutfile2 = fopen ( a, "w" ) ) == NULL )
    printf ( "Could not export data\n" );
  else {
    fprintf ( ptrOutfile2, "Gen_%s, PopSize_%s, He_%s, SDHe_%s,
AllelicDiv_%s, SDAllelicDiv_%s, Adi_%s, AdiSD_%s, dispersal_%s,
Density_%s, NBSizeAr_%s, NBSizeArSD_%s, NBSizeAhat_%s,
NBSizeAhatSD_%s\n", NameString, NameString, NameString, NameString,
NameString, NameString, NameString, NameString, NameString,
NameString, NameString, NameString, NameString, NameString );
    for ( i = 0; i < Y; i++ ) {
      pSize = PopSizeArray [ i ];
      D = (pSize/((double)(Rows*Cols)));
      Tdiv = TotalDiv [ i ];
      TdivSD = TotalDivSD [ i ];
      ADiv = TotalAllelicDiv [ i ];
      ADivSD = TotalAllelicDivSD [ i ];
      Nb1 = NbSizeAr [ i ];
      Nb1SD = NbSizeArSD [ i ];
      Nb2 = NbSizeAhat [ i ];
      Nb2SD = NbSizeAhatSD [ i ];
      ADI = AdiArray [ i ];
      ADISD = AdiSDArray [ i ];
      d = MeanDispersal [ i ];
      fprintf( ptrOutfile2,
"%d,%d,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f\n", i+1, pSize, Tdiv,
TdivSD, ADiv, ADivSD, ADI, ADISD, d, D, Nb1, Nb1SD, Nb2, Nb2SD );
    }
  }
}

void CalculateAlleleDistribution ( int N, int L, int Pop, int Y )
/*Calculates the statistic Adi, the distribution of shared alleles
with geographical distance.*/
{

  int pc;
  int i,j,k,l;
  int row, col;
  int NumAlleles;
  int testAllele;
  int A1, A2, B1, B2;
  int Testcase;
  int ctr;

  float SharedDistance;
  float Distance;
  float TotalDistance;

  //FILE *ptrTestFile;
  //FILE *ptrTestFile2;
  /*
  if ( ( ptrTestFile = fopen ( "test1.csv", "w" ) ) == NULL )
    printf ( "Could not open test1\n" );
  if ( ( ptrTestFile2 = fopen ( "test2.csv", "w" ) ) == NULL )
    printf ( "Could not open test2\n" );
  */
  Adi = 0;
  SharedDistance = 0;
  TotalDistance = 0;
  ctr = 0;
```

```
  //printf ( "Number of Loci is %d\n", L );
  for ( i = 0; i < L; i++ ) {
    NumAlleles = LocAlleleNumbers [ i ];
    for ( j = 0; j < NumAlleles; j++ ) {
      testAllele = Alleles[ j ][ i ];
      pc = 0;
      for ( k = 0; k < N-1; k++ ) {
    A1 = Data1[ k ][ i ];
    A2 = Data2[ k ][ i ];
    for ( l = k+1; l < N; l++ ) {
      //printf ( "ctr is %d\n", ctr );
      Distance = DistanceArray [ pc ];
      //fprintf ( ptrTestFile2, "%f\n", Distance );
      //printf ( "Distance is %f\n", Distance );
      B1 = Data1[ l ][ i ];
      B2 = Data2[ l ][ i ];
      Testcase = 0;
      if ( A1 == testAllele )
        Testcase += 1;
      if ( A2 == testAllele )
        Testcase += 1;
      if ( B1 == testAllele )
        Testcase += 1;
      if ( B2 == testAllele )
        Testcase += 1;
      if ( Testcase == 4 ) { //All four alleles in both individuals
identical and thus homozygotes
        //printf ( "Shared 4\n" );
        SharedDistance = SharedDistance + 2*Distance;
        //fprintf ( ptrTestFile, "%f\n", 2*Distance );
        if ( WriteSharedAlleleDistances == 1 ) {
          SharedDistances[ ctr ][ Y ][ Pop ] = Distance;
          ctr+=1;
        }
      }
      if ( Testcase == 3 ) {
        //printf ( "Shared 3\n" );
        SharedDistance = SharedDistance + 1.5*Distance;
        //fprintf ( ptrTestFile, "%f\n", 1.5*Distance );
        if ( WriteSharedAlleleDistances == 1 ) {
          SharedDistances[ ctr ][ Y ][ Pop ] = Distance;
          ctr+=1;
        }
      }
      if ( Testcase == 2 ) {
        if ( ( A1 == A2 ) || ( B1 == B2 ) ) {
          SharedDistance = SharedDistance;
        }
        else {
          //printf ( "Shared 2\n" );
          SharedDistance = SharedDistance + Distance;
          //fprintf ( ptrTestFile, "%f\n", Distance );
          if ( WriteSharedAlleleDistances == 1 ) {
        SharedDistances[ ctr ][ Y ][ Pop ] = Distance;
        ctr+=1;
          }
        }
      }
      pc+=1;
      TotalDistance = TotalDistance + 2*Distance;
    }
```

```
      }
    }
  }
  Adi = SharedDistance/TotalDistance;
  //printf ( "Adi is %f\n", Adi );
  //fclose ( ptrTestFile );
  //fclose ( ptrTestFile2 );
}

void CalculateDiversity ( int N, int L )
/*This procedure calculates heterozygosity from the sampled
population*/
{

    int i,j;
    int Row, Column;
    int Allele1, Allele2;

    H = 0;
    for ( i = 0; i < N; i++ ) {
    for ( j = 0; j < L; j++ ) {
        Row = SampledRows[i];
        Column = SampledColumns[i];
        Allele1 = Data[Row][Column][j][0];
        Allele2 = Data[Row][Column][j][1];
        if ( Allele1 != Allele2 ) {
          H+=1;
        }
    }
    }
    H = H/(N*L);
}

void CalculateAllelicDiversity ( int N, int L )
/*Calculates allelic diversity from sampled population*/
{
  int i;
  int NumAlleles, TotalAlleles;

  TotalAlleles = 0;
  for ( i = 0; i < L; i++ ) {
    NumAlleles = LocAlleleNumbers [ i ];
    TotalAlleles+=NumAlleles;
  }
  //printf ( "Total Alleles = %d\n", TotalAlleles );
  AllelicDiv = (float)( TotalAlleles/(float)(N*2*L) );
  //printf ( "Allelic Div is %f\n", AllelicDiv );
}


/*Implement a Ricker model with stochastic r to achieve population
cycles*/
int PopSizeCalc ( int Nt, float Ro, float var, int LatRows, int
LatCols )
{

  float Et;
  float R;
  int sizeP;
  int Ntplus1;
  int k;
```

```
  R = 0;
  Et = normal ( sqrt ( var ) );
  //printf ( "Et = %f\n", Et );
  R = Ro+Et;
  //printf ( "R is %f\n", R );
  k = LatRows*LatCols;
  //printf ( "k is %d\n", k );
  Ntplus1 = floor( Nt*exp( R*(1-Nt/k) ) );

  return Ntplus1;

}

void WritePopMatrix ( int LatRows, int LatCols, int L, int Y )
{
  int i, j, k;
  int Allele1, Allele2, Sex;

  fprintf ( ptrOutfile, "Year_%d\n", Y+1 );
  for ( i = 0; i < LatRows; i++ ) {
    for ( j = 0; j < LatCols; j++ ) {
      for ( k = 0; k < L; k++ ) {
    if ( k == 0 )
      Sex = sexData[ i ][ j ];
    Allele1 = Data[ i ][ j ][ k ][ 0 ];
    Allele2 = Data[ i ][ j ][ k ][ 1 ];
    fprintf( ptrOutfile, "%d\t%d\t%d\n", Allele1, Allele2, Sex );
      }
    }
  }
  fprintf( ptrOutfile, "\n" );
}

void Regression ( float X[], float Y1[], float Y2[], int n )
{
    int i;

    float x,y1,y2,xy1,xy2;
    float SumX,SumY1,SumY2;
    float AvgX,AvgY1,AvgY2;
    float SSX,SSY1,SSY2;
    float SSxy1, SSxy2;
    float ry1,ry2;
    float yint1,yint2;

    SumX = 0;
    SumY1 = 0;
    SumY2 = 0;
    x = 0;
    y1 = 0;
    y2 = 0;
    AvgX = 0;
    AvgY1 = 0;
    AvgY2 = 0;
    for ( i = 0; i < n-1; i++ ) {
    x = X[i];
    y1 = Y1[i];
    y2 = Y2[i];
    SumX+=x;
    SumY1+=y1;
```

```
    SumY2+=y2;
    }

    AvgX = SumX/n;
    AvgY1 = SumY1/n;
    AvgY2 = SumY2/n;

    SSX = 0;
    SSY1 = 0;
    SSY2 = 0;
    SSxy1 = 0;
    SSxy2 = 0;
    ry1 = 0;
    ry2 = 0;
    yint1 = 0;
    yint2 = 0;
    NbSizey1 = 0;
    NbSizey2 = 0;
    for ( i = 0; i < n-1; i++ ) {
    x = pow(( X[i] - AvgX ),2);
    y1 = pow(( Y1[i] - AvgY1 ),2);
    y2 = pow(( Y2[i] - AvgY2 ),2);
    SSX+=x;
    SSY1+=y1;
    SSY2+=y2;
    xy1 = (X[i]-AvgX)*(Y1[i]-AvgY1);
    xy2 = (X[i]-AvgX)*(Y2[i]-AvgY2);
    SSxy1+=xy1;
    SSxy2+=xy2;
    }
    ry1 = SSxy1/SSX;
    ry2 = SSxy2/SSX;
    yint1 = AvgY1 - ry1*AvgX;
    yint2 = AvgY2 - ry2*AvgX;
    NbSizey1 = 1/ry1;
    NbSizey2 = 1/ry2;
    //printf ( "NbSizey1 = %f; NbSizey2 = %f\n", NbSizey1, NbSizey2
);
}

void LogOfDistance( float Array[], int n )
{
    int i;

    float dist, logdist;

    LogDistance = ( float * ) calloc ( n, sizeof( float ) );
    if ( LogDistance == NULL ) {
    printf ( "LofDistance memory allocation failure\n" );
    }
    else {
    for ( i = 0; i < n-1; i++ ) {
        dist = Array[ i ];
        if ( dist == 0 ) {
        dist = 1;
        }
        logdist = log(dist);
        LogDistance[ i ] = logdist;

    }
    }
```

```
}

void CalculateAr ( int N, int L )
{
    int pairwiseCounter;
    int a,b,d,e,q,r;
    int NumAlleles;
    int counter;
    int TestAllele1;
    int Allele1, Allele2, Allele11, Allele12, Allele21, Allele22;
    int Row, Column;
    int Row1, Column1;
    int XSquare, YSquare;

    float X11, X12, X21, X22;
    float AvgXiu, AvgXiub, AvgXu;
    float Term1, Term2;
    float Denominator,Numerator, Denominator2, Numerator2;
    float SumDenominator, SumNumerator, SumDenominator2,
SumNumerator2;
    float SSb, SSw, SSw1;
    float Ar, Ahat;
    float Distance;

    DistanceArray = (float *) calloc ( (N*(N-1))/2, ( sizeof( float
)));
    if ( DistanceArray == NULL ) {
      printf( "DistanceArray memory allocation failure\n" );
    }

    ArArray = (float *) calloc ( ( N*(N-1))/2, ( sizeof( float )));
    if ( ArArray == NULL ) {
      printf( "ArArray memory allocation failure\n" );
    }

    AhatArray = (float *) calloc ( ( N*(N-1))/2, ( sizeof( float )));
    if ( AhatArray == NULL ) {
      printf( "AhatArray memory allocation failure\n" );
    }

    LocusSSW = (int *) calloc ( L, ( sizeof(int)));
    if ( LocusSSW == NULL ) {
      printf( "LocusSSW memory failure\n" );
    }

    pairwiseCounter = 0;
    for ( q = 0; q < N-1; q++ ) {  /*Calculate Ar and Ahat for each
pair of individuals sampled*/
    for ( r = q+1; r < N; r++ ) {
        SumDenominator = 0;
        SumNumerator = 0;
        SumDenominator2 = 0;
        SumNumerator2 = 0;
        for ( d = 0; d < L; d++ ) {
        Numerator = 0;
        Denominator = 0;
        Numerator2 = 0;
        Denominator2 = 0;
        NumAlleles = LocAlleleNumbers[d];
        if ( ( q == 0) && ( r == 1) ) { /*Calculate the SSw over all
pairwise comparisons only once for each locus*/
```

```
            SSw1 = 0;
            counter = 0;
            for ( a = 0; a < N-1; a++ ) {
            for ( b = a+1; b < N; b++ ) {
                Allele11 = Data1[a][d]; /*Allele 1 ind 1 of pairwise
comp*/
                Allele12 = Data2[a][d]; /*Allele 2 ind 1 of pairwise
comp*/
                Allele21 = Data1[b][d]; /*Allele 1, ind 2 of pairwise
comp*/
                Allele22 = Data2[b][d]; /*Allele 2, ind 2 of pairwise
comp*/
                for ( e = 0; e < NumAlleles; e++ ) {
                Term1 = 0;
                TestAllele1 = Alleles[e][d];
                if ( TestAllele1 == Allele11 ) {
                    X11 = 1;
                }
                else {
                    X11 = 0;
                }
                //printf( "allele %d X11 = %f\n", e, X11 );
                if ( TestAllele1 == Allele12 ) {
                    X12 = 1;
                }
                else {
                    X12 = 0;
                }
                //printf( "allele %d X12 = %f\n", e, X12 );
                if ( TestAllele1 == Allele21 ) {
                    X21 = 1;
                }
                else {
                    X21 = 0;
                }
                //printf( "allele %d X21 = %f\n", e, X21 );
                if ( TestAllele1 == Allele22 ) {
                    X22 = 1;
                }
                else {
                    X22 = 0;
                }
                //printf( "allele %d X22 = %f\n", e, X22 );
                AvgXiu = (X11+X12)/2;
                //printf( "allele %d AvgXiu = %f\n", e, AvgXiu );
                AvgXiub = (X21+X22)/2;
                //printf( "allele %d AvgXiub = %f\n", e, AvgXiub );
                    Term1 = pow((X11-AvgXiu),2) + pow((X12-AvgXiu),2)
+ pow((X21-AvgXiub),2) + pow((X22-AvgXiub),2);
                    SSw1+=Term1; /*Keep adding over all alleles and
pairwise comparisons*/
                }
                counter+=1;
            }
            }
            LocusSSW[d] = SSw1;
        }
        SSw1 = LocusSSW[d];
        Denominator = 2*SSw1;
        Denominator2 = SSw1;
/*
```

```
        if ( ( q == 0) && ( r == 1) ) {
            printf( "locus %d, SSW is %f\n", d, SSw1 );
        }
*/
        Allele11 = Data1[q][d];
        Allele12 = Data2[q][d];
        Allele21 = Data1[r][d];
        Allele22 = Data2[r][d];
        SSw = 0;
        SSb = 0;
        //printf( "NumAlleles is %d\n", NumAlleles );
        for ( e = 0; e < NumAlleles; e++ ) {
            TestAllele1 = Alleles[e][d];
            Term1 = 0;
            Term2 = 0;
            //printf( "Test allele is %d\n", TestAllele1 );
            if ( TestAllele1 == Allele11 ) {
            X11 = 1;
            }
            else {
            X11 = 0;
            }
            if ( TestAllele1 == Allele12 ) {
            X12 = 1;
            }
            else {
            X12 = 0;
            }
            if ( TestAllele1 == Allele21 ) {
            X21 = 1;
            }
            else {
            X21 = 0;
            }
            if ( TestAllele1 == Allele22 ) {
            X22 = 1;
            }
            else {
            X22 = 0;
            }
            AvgXu = (X11 + X12 + X21 + X22)/4;
            //printf ( "allele %d: AvgXu = %f\n", e, AvgXu );
            AvgXiu =  (X11+X12)/2;
            //printf ( "allele %d: AvgXiu = %f\n", e, AvgXiu );
            AvgXiub = (X21+X22)/2;
            //printf ( "allele %d: AvgXiub = %f\n", e, AvgXiub );
            Term1 = pow((X11-AvgXiu),2) + pow((X12-AvgXiu),2) +
pow((X21-AvgXiub),2) + pow((X22-AvgXiub),2);
            //printf ( "allele %d: term1 = %f\n", e, Term1 );
            SSw+=Term1;
            Term2 = 2*(pow((AvgXiu-AvgXu),2) + pow((AvgXiub-
AvgXu),2));
            //printf ( "allele %d: term2 = %f\n", e, Term2 );
            SSb+=Term2;
        }
        Numerator = ( 2*(SSb)-SSw)*((N*(N-1))/2);
        Numerator2 = ( SSb*((N*(N-1))/2) );
        SumDenominator+=Denominator;
        SumDenominator2+=Denominator2;
        SumNumerator+=Numerator;
        SumNumerator2+=Numerator2;
```

```
        //printf( "Locus %d: Numerator = %f, Denominator = %f\n", d,
Numerator, Denominator );
        //printf( "Locus %d: Numerator2 = %f, Denominator2 = %f\n",
d, Numerator2, Denominator2 );

        }
        Ar = SumNumerator/SumDenominator;
        Ahat = (SumNumerator2/SumDenominator2) - 0.5;

            Row = DataRow[q];
        Column = DataCol[q];
        Row1 = DataRow[r];
        Column1 = DataCol[r];
        ArArray[pairwiseCounter] = Ar;
        AhatArray[pairwiseCounter] = Ahat;

        if ( Row >= Row1 ) {
        YSquare = pow((Row-Row1),2);
        }
        else
        YSquare = pow((Row1-Row),2);
        if ( Column >= Column1 ) {
        XSquare = pow((Column-Column1),2);
        }
        else
        XSquare = pow((Column1-Column),2);
        Distance = sqrt(XSquare+YSquare);
        DistanceArray[pairwiseCounter] = Distance;
        pairwiseCounter+=1;

    }
    }
}


void CountAlleles( int N, int L )
{
  int i,q,k,l,p;
  int Allele1, Allele2;
  int match1, match2;
  int TestAllele1;

  Alleles = calloc ( N*2, ( sizeof( int *)));
  if ( Alleles == NULL ) {
    printf( "Alleles 1st dimension memory allocation failure\n" );
  }
  for ( l = 0; l < N*2; l++ ) {
    Alleles[l] = calloc ( L, sizeof( int ) );
    if ( Alleles[l] == NULL ) {
      printf( "Alleles 2nd dimension memory allocation failure\n" );
    }
  }

  LocAlleleNumbers = ( int * ) calloc ( L, sizeof ( int ) );
  if ( LocAlleleNumbers == NULL )
    printf ( "LocAlleleNumbers memory allocation failure\n" );

  for ( i = 0; i < L; i++ ) {
    p = 0;
    for ( k = 0; k < N; k++ ) {
```

```
      Allele1 = Data1[k][i];
      Allele2 = Data2[k][i];
      if ( k == 0 ) {
    if ( Allele1 == Allele2 ) {
      Alleles[p][i] = Allele1;
      p+=1;
    }
    else {
      Alleles[p][i] = Allele1;
      p+=1;
      Alleles[p][i] = Allele2;
      p+=1;
    }
      }
      else {
    match1 = 0;
    match2 = 0;
    if ( Allele1 != Allele2 ) {
      for ( q = 0; q < p; q++ ) {
        TestAllele1 = Alleles[q][i];
        if ( TestAllele1 == Allele1 ) {
          match1 = 1;
        }
        if ( TestAllele1 == Allele2 ) {
          match2 = 1;
        }
      }
      if ( match1 == 0) {
        Alleles[p][i] = Allele1;
        p+=1;
      }
      if ( match2 == 0) {
        Alleles[p][i] = Allele2;
        p+=1;
      }
    }
    else if ( Allele1 == Allele2 ) {
      for ( q = 0; q < p; q++ ) {
        TestAllele1 = Alleles[q][i];
        if ( TestAllele1 == Allele1 ) {
          match1 = 1;
        }
      }
      if ( match1 == 0 ) {
        Alleles[p][i] = Allele1;
        p+=1;
      }
    }
      }
    }
    LocAlleleNumbers[i] = p;
    }
}


void SamplePopulationII ( int N, int NumRows, int NumCols, int L )
{

  int i,j,k,l,m,n;
  int RandomRow;
  int RandomCol;
```

```
int test, testAllele;
int testRow, testColumn;
int match;
int Allele1, Allele2;
int reps;

SampledRows = (int *) calloc ( N, ( sizeof( int ) ) );
if ( SampledRows == NULL ) {
  printf( "SampledRows memory failure\n" );
}

SampledColumns = (int *) calloc ( N, ( sizeof( int ) ) );
if ( SampledColumns == NULL ) {
  printf( "SampledColumns memory failure\n" );
}

DataRow = ( int * ) calloc ( N, ( sizeof ( int ) ) );
if ( DataRow == NULL ) {
  printf( "Data row memory allocation failure\n" );
}

DataCol = ( int * ) calloc (N, ( sizeof ( int ) ) );
if ( DataCol == NULL ) {
  printf( "Data col memory allocation failure\n" );
}

Data1 = calloc ( N, ( sizeof(int *)));
if ( Data1 == NULL ) {
  printf( "Data1 1st dimension memory allocation failure\n" );
}
for ( l = 0; l < N; l++ ) {
  Data1[l] = calloc ( L, sizeof( int ) );
  if ( Data1[l] == NULL ) {
    printf ( "Data1 2nd dimension memory allocation failure\n" );
  }
}

Data2 = calloc ( N, ( sizeof(int *)));
if ( Data2 == NULL ) {
  printf( "Data2 1st dimension memory allocation failure\n" );
}
for ( l = 0; l < N; l++ ) {
  Data2[l] = calloc ( L, sizeof( int ) );
  if ( Data2[l] == NULL ) {
    printf ( "Data2 2nd dimension memory allocation failure\n" );
  }
}

ActualN = 0;
reps = 0;
n = 0;
while ( ( ActualN < N ) && ( reps < 100*N) ) {
  test = 0;
  match = 1;
  if ( ActualN == 0 ) {
    while ( test == 0 ) {
  RandomRow = rand() % NumRows;
  RandomCol = rand() % NumCols;
  testAllele = Data[RandomRow][RandomCol][0][0];
  if ( testAllele > 0 ) {
    test = 1;
```

```
      SampledRows[ ActualN ] = RandomRow;
      SampledColumns[ ActualN ] = RandomCol;
      ActualN+=1;
    }
      }
    }
    else {
      while ( match == 1 ) {
    RandomRow = rand() % NumRows;
    RandomCol = rand() % NumCols;
    testAllele = Data[ RandomRow ][ RandomCol ][ 0 ][ 0 ];
    if ( testAllele > 0 ) {
      test = 1;
    }
    if ( test == 1 ) {
      match = 0;
      for ( j = 0; j < ActualN-1; j++ ) {
        testRow = SampledRows[ j ];
        testColumn = SampledColumns[ j ];
        if ( ( testRow == RandomRow ) && ( testColumn == RandomCol )
) {
          match = 1;
        }
      }
      if ( match == 0 ) {
        SampledRows[ ActualN ] = RandomRow;
        SampledColumns [ ActualN ] = RandomCol;
        ActualN+=1;
      }
    }
      }
    }

    for ( l = 0; l < L; l++ ) {
      Allele1 = Data [ RandomRow ][ RandomCol ][ l ][ 0 ];
      Allele2 = Data [ RandomRow ][ RandomCol ][ l ][ 1 ];
      //printf ( "Allele1 is %d\n", Allele1 );
      //printf ( "Allele2 is %d\n", Allele2 );
      Data1[ ActualN-1 ][l] = Allele1;
      Data2[ ActualN-1 ][l] = Allele2;
    }
    DataRow [ ActualN-1 ] = RandomRow;
    DataCol [ ActualN-1 ] = RandomCol;
    reps+=1;
  }
}

void SamplePopulation ( int N, int NumRows, int NumCols, int L, int
extent, int sigma )
{
    int i,j,k,l,m,n;
    int RandomRow;
    int RandomCol;
    int test, testAllele;
    int testRow, testColumn;
    int match;
    int Allele1, Allele2;
    int CentroidRow, CentroidCol;
    int DistanceR, DistanceC;
    int reps;
    int e;
```

```
    float DirectionR, DirectionC;


    e = floor( 0.5*extent*sigma );

    SampledRows = (int *) calloc ( N, ( sizeof( int ) ) );
    if ( SampledRows == NULL ) {
      printf( "SampledRows memory failure\n" );
    }

    SampledColumns = (int *) calloc ( N, ( sizeof( int ) ) );
    if ( SampledColumns == NULL ) {
    printf( "SampledColumns memory failure\n" );
    }

    DataRow = ( int * ) calloc ( N, ( sizeof ( int ) ) );
    if ( DataRow == NULL ) {
      printf( "Data row memory allocation failure\n" );
    }

    DataCol = ( int * ) calloc (N, ( sizeof ( int ) ) );
    if ( DataCol == NULL ) {
      printf( "Data col memory allocation failure\n" );
    }

    Data1 = calloc ( N, ( sizeof(int *)));
    if ( Data1 == NULL ) {
      printf( "Data1 1st dimension memory allocation failure\n" );
    }
    for ( l = 0; l < N; l++ ) {
      Data1[l] = calloc ( L, sizeof( int ) );
      if ( Data1[l] == NULL ) {
    printf ( "Data1 2nd dimension memory allocation failure\n" );
      }
    }

    Data2 = calloc ( N, ( sizeof(int *)));
    if ( Data2 == NULL ) {
      printf( "Data2 1st dimension memory allocation failure\n" );
    }
    for ( l = 0; l < N; l++ ) {
      Data2[l] = calloc ( L, sizeof( int ) );
      if ( Data2[l] == NULL ) {
    printf ( "Data2 2nd dimension memory allocation failure\n" );
      }
    }

    ActualN = 0;
    reps = 0;
    n = 0;
    while ( ( ActualN < N ) && ( reps < 100*N) ) { //possible that
may not find N samples within the sampling range specified by extent.
      test = 0;
      match = 1;
      if ( ActualN == 0 ) {
        while ( test == 0 ) {
          RandomRow = rand() % NumRows;
          RandomCol = rand() % NumCols;
          testAllele = Data[RandomRow][RandomCol][0][0];
          if ( testAllele > 0 ) {
```

```
          test = 1;
          CentroidRow = RandomRow;
          CentroidCol = RandomCol;
          SampledRows[ ActualN ] = RandomRow;
          SampledColumns[ ActualN ] = RandomCol;
          ActualN+=1;
            }
          }
        }
        else {
          while ( match == 1 ) {
            DistanceR = floor( (rand() % ( e )) + 1 );
            DistanceC = floor( (rand() % ( e )) + 1 );
            DirectionR = ( rand()/((double)RAND_MAX+1) );
            DirectionC = ( rand()/((double)RAND_MAX+1) );

            if ( DirectionR > 0.5 ) {
          RandomRow = CentroidRow+DistanceR;
          if ( RandomRow > NumRows-1 )
            RandomRow = CentroidRow-DistanceR;
            }
            else {
          RandomRow = CentroidRow-DistanceR;
          if ( RandomRow < 0 )
            RandomRow = CentroidRow+DistanceR;
            }

            if ( DirectionC > 0.5 ) {
          RandomCol = CentroidCol+DistanceC;
          if ( RandomCol > NumCols-1 )
            RandomCol = CentroidCol-DistanceC;
            }
            else {
          RandomCol = CentroidCol-DistanceC;
          if ( RandomCol < 0 )
            RandomCol = CentroidCol+DistanceC;
            }
            testAllele = Data[ RandomRow ][ RandomCol ][ 0 ][ 0 ];
            if ( testAllele > 0 ) {
          test = 1;
            }
            if ( test == 1 ) {
          match = 0;
          for ( j = 0; j < ActualN-1; j++ ) {
            testRow = SampledRows[ j ];
            testColumn = SampledColumns[ j ];
            if ( ( testRow == RandomRow ) && ( testColumn == RandomCol
) ) {
              match = 1;
            }
          }
          if ( match == 0 ) {
            SampledRows[ ActualN ] = RandomRow;
            SampledColumns [ ActualN ] = RandomCol;
            ActualN+=1;
          }
            }
          }
        }
        for ( l = 0; l < L; l++ ) {
          Allele1 = Data [ RandomRow ][ RandomCol ][ l ][ 0 ];
```

```
        Allele2 = Data [ RandomRow ][ RandomCol ][ l ][ 1 ];
        Data1[ ActualN-1 ][l] = Allele1;
        Data2[ ActualN-1 ][l] = Allele2;
      }
      DataRow [ ActualN-1 ] = RandomRow;
      DataCol [ ActualN-1 ] = RandomCol;
      reps+=1;
    }
}


void WriteDataMatrix ( int LatRows, int LatCols, int Loci )
{

    int i,j,k,l;
    int dataVal, dataVal2, sexVal;

    for ( i = 0; i < LatRows; i++ ) {
      for ( j = 0; j < LatCols; j++ ) {
        sexVal = sexTemp[i][j];
    sexData[i][j] = sexVal;
    for ( k = 0; k < Loci; k ++ ) {
      dataVal = dataTemp[i][j][k][0];
      dataVal2 = dataTemp[i][j][k][1];
      Data[i][j][k][0] = dataVal;
      Data[i][j][k][1] = dataVal2;
        }
      }
    }
    return;
}



void Reproduction ( int PopSize, int LatRows, int LatCols, int d, int
L, int year )
{

  int j, m;
  int occupied;
  int RandomRow1, RandomRow2;
  int RandomCol1, RandomCol2;
  int Test;
  int male, reps;
  int RandomMaleX, RandomMaleY;
  int empty;
  int counter;
  int RandomOffspringX, RandomOffspringY;
  int OffspringX, OffspringY;
  int FemaleAllele1, FemaleAllele2;
  int MaleAllele1, MaleAllele2;
  int RandomAllele1, RandomAllele2;
  int Sex;
  int XDistance, YDistance;

  float DirectionX, DirectionY;
  float MuRate;
  float RandomMu;
  float RandomMutate;
  float RandomSex;
  float Random;
```

```
  float DispersalDistance;
  float AvgDispersal;
  float *DispersalDistanceArray;

  DispersalDistanceArray = ( float * ) calloc ( PopSize, sizeof (
float ) );
  if ( DispersalDistanceArray == NULL )
    printf ( "DispersalDistanceArray memory allocation failure\n" );

  j = 0;
  while ( ( j < PopSize ) && ( j < LatRows*LatCols ) ) {
    occupied = 0;
    while ( occupied == 0 ) { /*Find a random female*/
      RandomRow1 = floor( rand() % LatRows );
      RandomCol1 = floor( rand() % LatCols );
      Test = Data [ RandomRow1 ][ RandomCol1 ][ 0 ][ 0 ];
      if ( Test != 0 ) {
    occupied = 1;
    Sex = sexData [ RandomRow1 ] [ RandomCol1 ];
    if ( Sex == 2 )
      occupied = 1;
    else
      occupied = 0;
      }
    }

    /*Find a random male within d units of the female*/
    male = 0;
    reps = 0;
    while ( ( male == 0) && ( reps < 1000) ) {
      DirectionX = ( rand()/((double)RAND_MAX+1) );
      DirectionY = ( rand()/((double)RAND_MAX+1) );
      RandomMaleX = floor( (rand() % d ) + 1);
      RandomMaleY = floor( (rand() % d ) + 1);
      if ( DirectionX > 0.5 ) {
    RandomRow2 = RandomRow1+RandomMaleX;
    if ( RandomRow2 > LatRows-1 ) {
      RandomRow2 = RandomRow1-RandomMaleX;
    }
      }
      else {
    RandomRow2 = RandomRow1-RandomMaleX;
    if ( RandomRow2 < 0 ) {
      RandomRow2 = RandomRow1+RandomMaleX;
    }
      }

      if ( DirectionY > 0.5 ) {
    RandomCol2 = RandomCol1+RandomMaleY;
    if ( RandomCol2 > LatCols-1 ) {
      RandomCol2 = RandomCol1-RandomMaleY;
    }
      }
      else {
    RandomCol2 = RandomCol1-RandomMaleY;
    if ( RandomCol2 < 0 ) {
      RandomCol2 = RandomCol1+RandomMaleY;
    }
      }
      Sex = sexData [ RandomRow2 ][ RandomCol2 ];
      if ( Sex == 1 )
```

```
    male = 1;
      reps+=1;
    }

  /*Find a new location for the single progeny*/
    if ( male == 1 ) {
      empty = 0;
      counter = 0;
      while ( ( empty == 0 ) && ( counter < 10000 ) ) {
    DirectionX = ( rand()/((double)RAND_MAX+1) );
    DirectionY = ( rand()/((double)RAND_MAX+1) );
    RandomOffspringX = floor(( rand() % d ) + 1);
    RandomOffspringY = floor(( rand() % d ) + 1);

    if ( DirectionX > 0.5 ) {
      OffspringX = RandomRow1+RandomOffspringX;
      if ( OffspringX > LatRows-1 ) {
        OffspringX = RandomRow1-RandomOffspringX;
      }
    }
    else {
      OffspringX = RandomRow1-RandomOffspringX;
      if ( OffspringX < 0 ) {
        OffspringX = RandomRow1+RandomOffspringX;
      }
    }

    if ( DirectionY > 0.5 ) {
      OffspringY = RandomCol1+RandomOffspringY;
      if ( OffspringY > LatCols-1 ) {
        OffspringY = RandomCol1-RandomOffspringY;
      }
    }
    else {
      OffspringY = RandomCol1-RandomOffspringY;
      if ( OffspringY < 0 ) {
        OffspringY = RandomCol1+RandomOffspringY;
      }
    }
    Test = dataTemp[OffspringX][OffspringY][0][0];
    if ( Test == 0 ) {
      empty = 1;
    }
    counter+=1;
      }

      if ( RandomRow1 >= OffspringX )
    XDistance = RandomRow1 - OffspringX;
      else
    XDistance = OffspringX-RandomRow1;

      if ( RandomCol1 >= OffspringY )
    YDistance = RandomCol1 - OffspringY;
      else
    YDistance = OffspringY-RandomCol1;
      DispersalDistance = sqrt( pow( XDistance, 2 ) + pow( YDistance,
2 ) );
      if ( WriteDispersalFile == 1 )
        fprintf ( ptrDispersalFile, "%f\n", DispersalDistance );
      DispersalDistanceArray [ j ] = DispersalDistance;
```

```
      RandomSex = rand()/((double)RAND_MAX+1);
      if ( RandomSex > 0.5 )
    Sex = 1;
      else
    Sex = 2;

      sexTemp [ OffspringX ][ OffspringY ] = Sex;

  /*Generate Alleles, through descent and mutation, and write to
dataTemp*/
      for ( m = 0; m < L; m++ ) {
    MuRate = MutationRate[m];
    FemaleAllele1 = Data[RandomRow1][RandomCol1][m][0];
    FemaleAllele2 = Data[RandomRow1][RandomCol1][m][1];
    MaleAllele1 = Data[RandomRow2][RandomCol2][m][0];
    MaleAllele2 = Data[RandomRow2][RandomCol2][m][1];

    RandomAllele1 = rand()/((double)RAND_MAX+1);
    RandomAllele2 = rand()/((double)RAND_MAX+1);
    if ( RandomAllele1 > 0.5 ) { /*choose allele1 from female*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          FemaleAllele1 = FemaleAllele1+1;
      }
        else if ( RandomMutate <= 0.5 ) {
          FemaleAllele1 = FemaleAllele1-1;
          if ( FemaleAllele1 <= 0 ) {
      FemaleAllele1 = FemaleAllele1+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][0] = FemaleAllele1;
    }
    else { /*choose allele2 from female*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          FemaleAllele2 = FemaleAllele2+1;
        }
        else if ( RandomMutate <= 0.5 ) {
          FemaleAllele2 = FemaleAllele2-1;
          if ( FemaleAllele2 <= 0 ) {
      FemaleAllele2 = FemaleAllele2+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][0] = FemaleAllele2;
    }
    if ( RandomAllele2 > 0.5 ) { /*choose allele1 from male*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          MaleAllele1 = MaleAllele1+1;
        }
        else if ( RandomMutate <= 0.5 ) {
          MaleAllele1 = MaleAllele1-1;
          if ( MaleAllele1 <= 0 ) {
```

```
        MaleAllele1 = MaleAllele1+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][1] = MaleAllele1;
    }
    else { /*choose allele2 from male*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          MaleAllele2 = MaleAllele2+1;
        }
        else if ( RandomMutate <= 0.5 ) {
          MaleAllele2 = MaleAllele2-1;
          if ( MaleAllele2 <= 0 ) {
        MaleAllele2 = MaleAllele2+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][1] = MaleAllele2;
    }
      }
      j+=1;
    }
    if ( j == ( (LatRows*LatCols)-1 ) ) {
      Random = rand()/((double)RAND_MAX+1);
      NodesOccupied = floor( LatRows*LatCols*Random );
    }
  }

  AvgDispersal = mean ( DispersalDistanceArray, PopSize );
  MeanDispersal [ year ] = AvgDispersal;
  free ( DispersalDistanceArray );

}

void ReproductionII ( int PopSize, int LatRows, int LatCols, int dp,
float v, int L, int year )
{

  int j, m;
  int d;
  int occupied;
  int RandomRow1, RandomRow2;
  int RandomCol1, RandomCol2;
  int Test;
  int male, reps;
  int RandomMaleX, RandomMaleY;
  int empty;
  int counter;
  int RandomOffspringX, RandomOffspringY;
  int OffspringX, OffspringY;
  int FemaleAllele1, FemaleAllele2;
  int MaleAllele1, MaleAllele2;
  int RandomAllele1, RandomAllele2;
  int Sex;
  int XDistance, YDistance;

  float DirectionX, DirectionY;
  float MuRate;
```

```
  float RandomMu;
  float RandomMutate;
  float RandomSex;
  float Random;
  float DispersalDistance;
  float AvgDispersal;
  float *DispersalDistanceArray;
  float FloatVal;

  DispersalDistanceArray = ( float * ) calloc ( PopSize, sizeof (
float ) );
  if ( DispersalDistanceArray == NULL )
    printf ( "DispersalDistanceArray memory allocation failure\n" );

  j = 0;
  while ( ( j < PopSize ) && ( j < LatRows*LatCols ) ) {
    occupied = 0;
    while ( occupied == 0 ) { /*Find a random female*/
      RandomRow1 = floor( rand() % LatRows );
      RandomCol1 = floor( rand() % LatCols );
      Test = Data [ RandomRow1 ][ RandomCol1 ][ 0 ][ 0 ];
      if ( Test != 0 ) {
    occupied = 1;
    Sex = sexData [ RandomRow1 ] [ RandomCol1 ];
    if ( Sex == 2 )
      occupied = 1;
    else
      occupied = 0;
      }
    }

    //Draw a maximum dispersal distance from a normal distribution
with mean dp, var v
    FloatVal = 0;
    while ( FloatVal <= 0 ) {
      FloatVal = ( ( normal ( sqrt ( v ) ) ) + dp );
    }
    d = floor( FloatVal + 1 ); //for Random normal values of between
0 and 1 we add 1

    /*Find a random male within d units of the female*/
    male = 0;
    reps = 0;
    while ( ( male == 0) && ( reps < 1000) ) {
      DirectionX = ( rand()/((double)RAND_MAX+1) );
      DirectionY = ( rand()/((double)RAND_MAX+1) );
      RandomMaleX = floor( (rand() % d ) + 1);
      RandomMaleY = floor( (rand() % d ) + 1);
      if ( DirectionX > 0.5 ) {
    RandomRow2 = RandomRow1+RandomMaleX;
    if ( RandomRow2 > LatRows-1 ) {
      RandomRow2 = RandomRow1-RandomMaleX;
    }
      }
      else {
    RandomRow2 = RandomRow1-RandomMaleX;
    if ( RandomRow2 < 0 ) {
      RandomRow2 = RandomRow1+RandomMaleX;
    }
      }
```

```
      if ( DirectionY > 0.5 ) {
    RandomCol2 = RandomCol1+RandomMaleY;
    if ( RandomCol2 > LatCols-1 ) {
      RandomCol2 = RandomCol1-RandomMaleY;
    }
      }
      else {
    RandomCol2 = RandomCol1-RandomMaleY;
    if ( RandomCol2 < 0 ) {
      RandomCol2 = RandomCol1+RandomMaleY;
    }
      }
      Sex = sexData [ RandomRow2 ][ RandomCol2 ];
      if ( Sex == 1 )
    male = 1;
      reps+=1;
    }

  /*Find a new location for the single progeny*/

    if ( male == 1 ) {

      //Draw a maximum dispersal distance from a normal distribution
with mean dp, var v
      FloatVal = 0;
      while ( FloatVal <= 0 ) {
        FloatVal = ( ( normal ( sqrt ( v ) ) ) + dp );
      }
      d = floor( FloatVal + 1 ); //for Random normal values of
between 0 and 1 we add 1

      empty = 0;
      counter = 0;
      while ( ( empty == 0 ) && ( counter < 10000 ) ) {
    DirectionX = ( rand()/((double)RAND_MAX+1) );
    DirectionY = ( rand()/((double)RAND_MAX+1) );
    RandomOffspringX = floor(( rand() % d ) + 1);
    RandomOffspringY = floor(( rand() % d ) + 1);

    if ( DirectionX > 0.5 ) {
      OffspringX = RandomRow1+RandomOffspringX;
      if ( OffspringX > LatRows-1 ) {
        OffspringX = RandomRow1-RandomOffspringX;
      }
    }
    else {
      OffspringX = RandomRow1-RandomOffspringX;
      if ( OffspringX < 0 ) {
        OffspringX = RandomRow1+RandomOffspringX;
      }
    }

    if ( DirectionY > 0.5 ) {
      OffspringY = RandomCol1+RandomOffspringY;
      if ( OffspringY > LatCols-1 ) {
        OffspringY = RandomCol1-RandomOffspringY;
      }
    }
    else {
      OffspringY = RandomCol1-RandomOffspringY;
      if ( OffspringY < 0 ) {
```

```
        OffspringY = RandomCol1+RandomOffspringY;
      }
    }
    Test = dataTemp[OffspringX][OffspringY][0][0];
    if ( Test == 0 ) {
      empty = 1;
    }
    counter+=1;
      }

      if ( RandomRow1 >= OffspringX )
    XDistance = RandomRow1 - OffspringX;
      else
    XDistance = OffspringX-RandomRow1;

      if ( RandomCol1 >= OffspringY )
    YDistance = RandomCol1 - OffspringY;
      else
    YDistance = OffspringY-RandomCol1;
      DispersalDistance = sqrt( pow( XDistance, 2 ) + pow( YDistance,
2 ) );
      if ( WriteDispersalFile == 1 )
    fprintf ( ptrDispersalFile, "%f\n", DispersalDistance );
      DispersalDistanceArray [ j ] = DispersalDistance;

      RandomSex = rand()/((double)RAND_MAX+1);
      if ( RandomSex > 0.5 )
    Sex = 1;
      else
    Sex = 2;

      sexTemp [ OffspringX ][ OffspringY ] = Sex;

  /*Generate Alleles, through descent and mutation, and write to
dataTemp*/
      for ( m = 0; m < L; m++ ) {
    MuRate = MutationRate[m];
    FemaleAllele1 = Data[RandomRow1][RandomCol1][m][0];
    FemaleAllele2 = Data[RandomRow1][RandomCol1][m][1];
    MaleAllele1 = Data[RandomRow2][RandomCol2][m][0];
    MaleAllele2 = Data[RandomRow2][RandomCol2][m][1];

    RandomAllele1 = rand()/((double)RAND_MAX+1);
    RandomAllele2 = rand()/((double)RAND_MAX+1);
    if ( RandomAllele1 > 0.5 ) { /*choose allele1 from female*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          FemaleAllele1 = FemaleAllele1+1;
      }
        else if ( RandomMutate <= 0.5 ) {
          FemaleAllele1 = FemaleAllele1-1;
          if ( FemaleAllele1 <= 0 ) {
        FemaleAllele1 = FemaleAllele1+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][0] = FemaleAllele1;
    }
    else { /*choose allele2 from female*/
```

```
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          FemaleAllele2 = FemaleAllele2+1;
        }
        else if ( RandomMutate <= 0.5 ) {
          FemaleAllele2 = FemaleAllele2-1;
          if ( FemaleAllele2 <= 0 ) {
        FemaleAllele2 = FemaleAllele2+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][0] = FemaleAllele2;
    }
    if ( RandomAllele2 > 0.5 ) { /*choose allele1 from male*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          MaleAllele1 = MaleAllele1+1;
        }
        else if ( RandomMutate <= 0.5 ) {
          MaleAllele1 = MaleAllele1-1;
          if ( MaleAllele1 <= 0 ) {
        MaleAllele1 = MaleAllele1+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][1] = MaleAllele1;
    }
    else { /*choose allele2 from male*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          MaleAllele2 = MaleAllele2+1;
        }
        else if ( RandomMutate <= 0.5 ) {
          MaleAllele2 = MaleAllele2-1;
          if ( MaleAllele2 <= 0 ) {
        MaleAllele2 = MaleAllele2+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][1] = MaleAllele2;
    }
      }
      j+=1;
    }
    if ( j == ( (LatRows*LatCols)-1 ) ) {
      Random = rand()/((double)RAND_MAX+1);
      NodesOccupied = floor( LatRows*LatCols*Random );
    }
  }

  AvgDispersal = mean ( DispersalDistanceArray, PopSize );
  MeanDispersal [ year ] = AvgDispersal;
  free ( DispersalDistanceArray );

}
```

```
void ReproductionIII ( int PopSize, int LatRows, int LatCols, int d1,
float v1, int d2, float v2, int L, int year )
{

  int d;
  int j, m;
  int occupied;
  int RandomRow1, RandomRow2;
  int RandomCol1, RandomCol2;
  int Test;
  int male, reps;
  int RandomMaleX, RandomMaleY;
  int empty;
  int counter;
  int RandomOffspringX, RandomOffspringY;
  int OffspringX, OffspringY;
  int FemaleAllele1, FemaleAllele2;
  int MaleAllele1, MaleAllele2;
  int RandomAllele1, RandomAllele2;
  int Sex;
  int XDistance, YDistance;

  float DirectionX, DirectionY;
  float MuRate;
  float RandomMu;
  float RandomMutate;
  float RandomSex;
  float Random;
  float DispersalDistance;
  float AvgDispersal;
  float *DispersalDistanceArray;
  float FloatVal;

  DispersalDistanceArray = ( float * ) calloc ( PopSize, sizeof (
float ) );
  if ( DispersalDistanceArray == NULL )
    printf ( "DispersalDistanceArray memory allocation failure\n" );

  j = 0;
  while ( ( j < PopSize ) && ( j < LatRows*LatCols ) ) {
    occupied = 0;
    while ( occupied == 0 ) { /*Find a random female*/
      RandomRow1 = floor( rand() % LatRows );
      RandomCol1 = floor( rand() % LatCols );
      Test = Data [ RandomRow1 ][ RandomCol1 ][ 0 ][ 0 ];
      if ( Test != 0 ) {
    occupied = 1;
    Sex = sexData [ RandomRow1 ] [ RandomCol1 ];
    if ( Sex == 2 )
      occupied = 1;
    else
      occupied = 0;
      }
    }

    FloatVal = 0;
    while ( FloatVal <= 0 ) {
      Random = rand()/((double)RAND_MAX+1);
      if ( Random >= 0.05 ) {
    FloatVal = ( ( normal ( sqrt ( v1 ) ) ) + d1 );
```

```
    }
    else if ( Random < 0.05 ) {
  FloatVal = ( ( normal ( sqrt ( v2 ) ) ) + d2 );
    }
  }
  d = floor( FloatVal + 1);


  /*Find a random male within d units of the female*/
  male = 0;
  reps = 0;
  while ( ( male == 0) && ( reps < 1000) ) {
    DirectionX = ( rand()/((double)RAND_MAX+1) );
    DirectionY = ( rand()/((double)RAND_MAX+1) );
    RandomMaleX = floor( (rand() % d ) + 1);
    RandomMaleY = floor( (rand() % d ) + 1);
    if ( DirectionX > 0.5 ) {
  RandomRow2 = RandomRow1+RandomMaleX;
  if ( RandomRow2 > LatRows-1 ) {
    RandomRow2 = RandomRow1-RandomMaleX;
  }
    }
    else {
  RandomRow2 = RandomRow1-RandomMaleX;
  if ( RandomRow2 < 0 ) {
    RandomRow2 = RandomRow1+RandomMaleX;
  }
    }

    if ( DirectionY > 0.5 ) {
  RandomCol2 = RandomCol1+RandomMaleY;
  if ( RandomCol2 > LatCols-1 ) {
    RandomCol2 = RandomCol1-RandomMaleY;
  }
    }
    else {
  RandomCol2 = RandomCol1-RandomMaleY;
  if ( RandomCol2 < 0 ) {
    RandomCol2 = RandomCol1+RandomMaleY;
  }
    }
    Sex = sexData [ RandomRow2 ][ RandomCol2 ];
    if ( Sex == 1 )
  male = 1;
    reps+=1;
  }

/*Find a new location for the single progeny*/
  if ( male == 1 ) {
    FloatVal = 0;
    while ( FloatVal <= 0 ) {
  Random = rand()/((double)RAND_MAX+1);
  if ( Random >= 0.05 ) {
    FloatVal = ( ( normal ( sqrt ( v1 ) ) ) + d1 );
  }
  else if ( Random < 0.05 ) {
    FloatVal = ( ( normal ( sqrt ( v2 ) ) ) + d2 );
  }
    }
    d = floor( FloatVal + 1);
```

```
      empty = 0;
      counter = 0;
      while ( ( empty == 0 ) && ( counter < 10000 ) ) {
    DirectionX = ( rand()/((double)RAND_MAX+1) );
    DirectionY = ( rand()/((double)RAND_MAX+1) );
    RandomOffspringX = floor(( rand() % d ) + 1);
    RandomOffspringY = floor(( rand() % d ) + 1);

    if ( DirectionX > 0.5 ) {
      OffspringX = RandomRow1+RandomOffspringX;
      if ( OffspringX > LatRows-1 ) {
        OffspringX = RandomRow1-RandomOffspringX;
      }
    }
    else {
      OffspringX = RandomRow1-RandomOffspringX;
      if ( OffspringX < 0 ) {
        OffspringX = RandomRow1+RandomOffspringX;
      }
    }

    if ( DirectionY > 0.5 ) {
      OffspringY = RandomCol1+RandomOffspringY;
      if ( OffspringY > LatCols-1 ) {
        OffspringY = RandomCol1-RandomOffspringY;
      }
    }
    else {
      OffspringY = RandomCol1-RandomOffspringY;
      if ( OffspringY < 0 ) {
        OffspringY = RandomCol1+RandomOffspringY;
      }
    }
    Test = dataTemp[OffspringX][OffspringY][0][0];
    if ( Test == 0 ) {
      empty = 1;
    }
    counter+=1;
      }

      if ( RandomRow1 >= OffspringX )
    XDistance = RandomRow1 - OffspringX;
      else
    XDistance = OffspringX-RandomRow1;

      if ( RandomCol1 >= OffspringY )
    YDistance = RandomCol1 - OffspringY;
      else
    YDistance = OffspringY-RandomCol1;
      DispersalDistance = sqrt( pow( XDistance, 2 ) + pow( YDistance,
2 ) );
      if ( WriteDispersalFile == 1 )
    fprintf ( ptrDispersalFile, "%f\n", DispersalDistance );
      DispersalDistanceArray [ j ] = DispersalDistance;

      RandomSex = rand()/((double)RAND_MAX+1);
      if ( RandomSex > 0.5 )
    Sex = 1;
      else
    Sex = 2;
```

```
      sexTemp [ OffspringX ][ OffspringY ] = Sex;

  /*Generate Alleles, through descent and mutation, and write to
dataTemp*/
      for ( m = 0; m < L; m++ ) {
    MuRate = MutationRate[m];
    FemaleAllele1 = Data[RandomRow1][RandomCol1][m][0];
    FemaleAllele2 = Data[RandomRow1][RandomCol1][m][1];
    MaleAllele1 = Data[RandomRow2][RandomCol2][m][0];
    MaleAllele2 = Data[RandomRow2][RandomCol2][m][1];

    RandomAllele1 = rand()/((double)RAND_MAX+1);
    RandomAllele2 = rand()/((double)RAND_MAX+1);
    if ( RandomAllele1 > 0.5 ) { /*choose allele1 from female*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          FemaleAllele1 = FemaleAllele1+1;
      }
        else if ( RandomMutate <= 0.5 ) {
          FemaleAllele1 = FemaleAllele1-1;
          if ( FemaleAllele1 <= 0 ) {
        FemaleAllele1 = FemaleAllele1+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][0] = FemaleAllele1;
    }
    else { /*choose allele2 from female*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          FemaleAllele2 = FemaleAllele2+1;
        }
        else if ( RandomMutate <= 0.5 ) {
          FemaleAllele2 = FemaleAllele2-1;
          if ( FemaleAllele2 <= 0 ) {
        FemaleAllele2 = FemaleAllele2+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][0] = FemaleAllele2;
    }
    if ( RandomAllele2 > 0.5 ) { /*choose allele1 from male*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          MaleAllele1 = MaleAllele1+1;
        }
        else if ( RandomMutate <= 0.5 ) {
          MaleAllele1 = MaleAllele1-1;
          if ( MaleAllele1 <= 0 ) {
        MaleAllele1 = MaleAllele1+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][1] = MaleAllele1;
    }
```

```
    else { /*choose allele2 from male*/
      RandomMu = rand()/((double)RAND_MAX+1);
      if ( MuRate > RandomMu ) {
        RandomMutate = rand()/((double)RAND_MAX+1);
        if ( RandomMutate > 0.5 ) {
          MaleAllele2 = MaleAllele2+1;
        }
        else if ( RandomMutate <= 0.5 ) {
          MaleAllele2 = MaleAllele2-1;
          if ( MaleAllele2 <= 0 ) {
        MaleAllele2 = MaleAllele2+2;
          }
        }
      }
      dataTemp[OffspringX][OffspringY][m][1] = MaleAllele2;
    }
      }
      j+=1;
    }
    if ( j == ( (LatRows*LatCols)-1 ) ) {
      Random = rand()/((double)RAND_MAX+1);
      NodesOccupied = floor( LatRows*LatCols*Random );
    }
  }

  AvgDispersal = mean ( DispersalDistanceArray, PopSize );
  MeanDispersal [ year ] = AvgDispersal;
  free ( DispersalDistanceArray );

}


void AssignMuRate ( float a, float b, int L )
{
    int i;
    float testAlpha, mu;
    float XGamma;

    testAlpha = a - floor( a );
    for ( i = 0; i < L; i++ ) {
    if ( testAlpha == 0 ) {
        XGamma = gammaMultiplication( a );
    }
    else if ( ( testAlpha != 0 ) && ( a > 0 ) && ( a < 1 ) ) {
        XGamma = gammaRejection( a );
    }
    else if ( ( testAlpha != 0) && ( a > 1 ) ) {
        XGamma = gammaTwoPart( a );
    }
    mu = ( XGamma/a )*( b );
    *(MutationRate+i) = mu;
    }
}

void InitialiseArrays ( int LatRows, int LatCols, int N, int L, int
Y, int Pops )
{
  int i, j, k, l;

  MutationRate = ( float *) calloc ( L, sizeof ( float ) );
  if ( MutationRate == NULL )
```

```
  printf( "Mutation rate memory allocation failure\n" );

MeanDispersal = ( float *) calloc ( Y, sizeof ( float ) );
if ( MeanDispersal == NULL ) {
  printf ( "MeanDispersal memory allocation failure\n" );
}

NbSizeAr = ( float * ) calloc ( Y, sizeof ( float ) );
if ( NbSizeAr == NULL ) {
  printf ( "NbSizeAr memory allocation failure\n" );
}

NbSizeArSD = ( float * ) calloc ( Y, sizeof ( float ) );
if ( NbSizeArSD == NULL ) {
  printf ( "NbSizeArSD memory allocation failure\n" );
}

NbSizeAhat = ( float * ) calloc ( Y, sizeof ( float ) );
if ( NbSizeAhat == NULL ) {
  printf ( "NbSizeAhat memory allocation failure\n" );
}

NbSizeAhatSD = ( float * ) calloc ( Y, sizeof ( float ) );
if ( NbSizeAhatSD == NULL ) {
  printf ( "NbSizeAhatSD memory allocation failure\n" );
}

PopSizeArray = ( int * ) calloc ( Y, sizeof ( int ) );
if ( PopSizeArray == NULL ) {
  printf ( "PopSizeArray memory allocation failure\n" );
}

TotalDiv = ( float * ) calloc (Y, ( sizeof ( float ) ) );
if ( TotalDiv == NULL ) {
  printf( "TotalDiv memory allocation failure\n" );
}

TotalDivSD = ( float * ) calloc (Y, sizeof ( float ) );
if ( TotalDivSD == NULL ) {
  printf( "TotalDivVar memory allocation failure\n" );
}

TotalAllelicDiv = ( float * ) calloc (Y, sizeof ( float ) );
if ( TotalAllelicDiv == NULL ) {
  printf( "TotalAllelicDiv memory allocation failure\n" );
}

TotalAllelicDivSD = ( float * ) calloc (Y, sizeof ( float ) );
if ( TotalAllelicDivSD == NULL ) {
  printf( "TotalAllelicDivSD memory allocation failure\n" );
}

AdiArray = ( float * ) calloc (Y, sizeof ( float ) );
if ( AdiArray == NULL ) {
  printf( "AdiArray memory allocation failure\n" );
}

AdiSDArray = ( float * ) calloc (Y, sizeof ( float ) );
if ( AdiSDArray == NULL ) {
  printf( "AdiSDArray memory allocation failure\n" );
}
```

```
  if ( WriteSharedAlleleDistances == 1 ) {
    SharedDistances = calloc ( ( ( L*N*(N-1) )/2 ), sizeof( float **
) );
    if ( SharedDistances == NULL ) {
      printf ( "1st dimension memory allocation failure:
SharedDistances\n" );
    }
    for ( i = 0; i < ( (L*N*(N-1))/2 ); i++ ) {
      SharedDistances[ i ] = calloc ( Y, sizeof( float *) );
      if ( SharedDistances[ i ] == NULL )
    printf ( "2nd dimension memory allocation failure:
SharedDistances\n" );
      for ( j = 0; j < Y; j++ ) {
    SharedDistances[ i ][ j ] = calloc ( Pops, sizeof( float ) );
    if ( SharedDistances[ i ][ j ] == NULL )
      printf( "3rd dimension memory allocation failure:
SharedDistances\n" );
      }
    }
  }

  Data = calloc ( LatRows, sizeof( int *** ) );
  if ( Data == NULL ) {
    printf ( "1st dimension memory allocation failure: Data\n" );
  }
  for ( i = 0; i < LatRows; i++ ) {
    Data[ i ] = calloc ( LatCols, sizeof( int ** ) );
    if ( Data[ i ] == NULL ) {
      printf ( "2nd dimension memory allocation failure: Data\n" );
    }
    for ( j = 0; j < LatCols; j++ ) {
      Data[ i ][ j ] = calloc ( L, sizeof( int * ) );
      if ( Data [ i ][ j ] == NULL ) {
    printf ( "3rd dimension memory allocation failure: Data\n" );
      }
      for ( k = 0; k < L; k++ ) {
    Data[ i ][ j ][ k ] = calloc ( 2, sizeof( int ) );
    if ( Data [ i ][ j ][ k ] == NULL ) {
      printf ( "4th dimension memory allocation failure: Data\n" );
    }
      }
    }
  }

  sexData = calloc ( LatRows, sizeof( int * ) );
  if ( sexData == NULL ) {
    printf( "1st dimension memory allocation failure: sexData\n" );
  }
  for ( i = 0; i < LatRows; i++ ) {
    sexData[ i ] = calloc( LatCols, sizeof( int ) );
    if ( sexData[ i ] == NULL ) {
      printf ( "2nd dimension memory allocation failure: sexData\n"
);
    }
  }



}
```

```
void ResetTempArrays ( int LatRows, int LatCols, int L )
{
    int i, j, k, l;

    for ( i = 0; i <= LatRows-1; i++ ) {
    for ( j = 0; j <= LatCols-1; j++ ) {
        sexTemp[i][j] = 0;
        for ( k = 0; k <= L-1; k++ ) {
        for ( l = 0; l < 2; l++ ) {
            dataTemp[i][j][k][l] = 0;
        }
        }
    }
    }
    return;
}

void CreateTempArrays ( int LatRows, int LatCols, int L )
{

  int i, j, k, l;

    dataTemp = calloc (LatRows, sizeof(int ***));
    if ( dataTemp == NULL ) {
      printf( "1st dimension memory allocation (dataTemp) failure\n"
);
    }
    for ( i = 0; i < LatRows; i++ ) {
      dataTemp[i] = calloc ( LatCols, sizeof(int **));
      if ( dataTemp[i] == NULL ) {
        printf( "2nd dimension memory allocation (dataTemp) failure
at %d\n", i );
      }
      for ( j = 0; j < LatCols; j++ ) {
        dataTemp[i][j] = calloc ( L, sizeof(int *));
    if ( dataTemp[i][j] == NULL ) {
          printf( "3rd dimension memory allocation (dataTemp) failure
at %d\t%d\n", i, j );
    }
    for ( k = 0; k < L; k++ ) {
          dataTemp[i][j][k] = calloc (2, sizeof(int));
          if ( dataTemp[i][j][k] == NULL ) {
            printf( "4th dimension memory allocation (dataTemp)
failure at %d\t%d\t%d\n", i, j, k );
          }
        }
      }
    }

    sexTemp = calloc (LatRows, sizeof(int *));
    if ( sexTemp == NULL ) {
      printf( "1st dimension memory allocation failure, sexTemp\n" );
    }
    for ( i = 0; i < LatRows; i++ ) {
      sexTemp[i] = calloc (LatCols, sizeof(int));
      if ( sexTemp[i] == NULL ) {
        printf( "2nd dimension memory allocation failure, sexTemp\n"
);
      }
    }
    return;
```

```
}


int SetStartingConditions ( int NumLoci, int LatticeRows, int
LatticeCols, int MeanNumAllelesPerLoc, float alleleAlpha, float het )
{
  int a,i,j,k,l,m,n,o;
  int NodeCol, NodeRow;
  int match;
  int testRow, testCol;
  int Alleles;
  int *LocAlleleNumbers;
  int Allele1, Allele2;
  int StartPopSize;

  float testAlpha;
  float XGamma;
  float RandomNum;

  LocAlleleNumbers = (int *) calloc ( NumLoci, ( sizeof(int)));
  if ( LocAlleleNumbers == NULL ) {
    printf( "LocAlleleNumbers memory failure\n" );
  }

  testAlpha = alleleAlpha - floor( alleleAlpha );
  for ( a = 0; a <= NumLoci-1; a++ ) {
    Alleles = 0;
    while ( ( Alleles < 2 ) ) {
      if ( testAlpha == 0 ) {
    XGamma = gammaMultiplication( alleleAlpha );
      }
      else if ( ( testAlpha != 0 ) && (alleleAlpha > 0 ) && (
alleleAlpha < 1 ) ) {
    XGamma = gammaRejection( alleleAlpha );
      }
      else if ( ( testAlpha != 0) && ( alleleAlpha > 1 ) ) {
    XGamma = gammaTwoPart( alleleAlpha );
      }
      Alleles = (XGamma/alleleAlpha)*(MeanNumAllelesPerLoc);
      *(LocAlleleNumbers+a) = Alleles;
    }
  }
  printf( "Number of Alleles per locus\n" );
  for ( l = 0; l < NumLoci; l++ ) {
    printf ("Loc%d\t", l+1 );
  }
  printf ( "\n");

  for ( l = 0; l <= NumLoci-1; l++ ) {
    Alleles = *(LocAlleleNumbers+l);
    printf( "%d\t", Alleles );
  }
  printf( "\n" );

  NodesOccupied = floor(LatticeRows*LatticeCols*percentOccupied);
  printf( "Nodes occupied is %d\n", NodesOccupied );

  NodeRows = (int *) calloc (NodesOccupied, ( sizeof(int)));
  if ( NodeRows == NULL ) {
    printf( "NodeRows memory failure\n" );
  }
```

```
  NodeCols = (int *) calloc (NodesOccupied, ( sizeof(int)));
  if ( NodeCols == NULL ) {
    printf( "NodeCols memory failure\n" );
  }

  for ( i = 0; i < NodesOccupied; i++ ) {
    match = 1;
    while ( match == 1) {
      NodeCol = ( rand() % (LatticeCols) );
      NodeRow = ( rand() % (LatticeRows) );
      if ( i == 0 ) {
    NodeRows[i] = NodeRow;
    NodeCols[i] = NodeCol;
    match = 0;
      }

      else if ( i > 0) {
    j = 0;
    match = 0;
    while ( ( j <= i-1) ) {
      testRow = NodeRows[j];
      testCol = NodeCols[j];
      if ( ( testRow == NodeRow) && ( testCol == NodeCol ) ) {
        match = 1;
      }
      j+=1;
    }

    if ( match == 0 ) {
      NodeRows[i] = NodeRow;
      NodeCols[i] = NodeCol;
    }
      }
    }

    for ( k = 0; k <= NumLoci-1; k++ ) {
      Alleles = *(LocAlleleNumbers+k);
      RandomNum = rand()/((double)RAND_MAX+1);
      if ( RandomNum <= het ) {
    Allele1 = (rand() % Alleles)+1;
    Allele2 = Allele1;
    while ( Allele1 == Allele2 ) {
      Allele2 = (rand() % Alleles)+1;
    }
    Data[NodeRow][NodeCol][k][0] = Allele1;
    Data[NodeRow][NodeCol][k][1] = Allele2;
      }
      else if ( RandomNum > het ) {
    Allele1 = (rand() % Alleles)+1;
    Data[NodeRow][NodeCol][k][0] = Allele1;
    Data[NodeRow][NodeCol][k][1] = Allele2;
      }
    }
  }
  return;
}

int AssignSex ( int NodesOccupied, int LatRows, int LatCols)
{
```

```
    int j;
    int NodeCol, NodeRow;
    int Sex;
    float RandomSex;
    int SizeMem;

    for ( j = 0; j < NodesOccupied; j++ ) {
    NodeRow = NodeRows[j];
    NodeCol = NodeCols[j];
        RandomSex = ( rand()/((double)RAND_MAX+1) );
      if ( RandomSex > 0.5 ) {
        Sex = 1; /*ie male*/
      }
      else {
        Sex = 2; /*ie female*/
      }
      sexData[NodeRow][NodeCol] = Sex;
    }
    return;
}

/* Gamma functions */
/*All gamma functions are from Ahrens & Dieter 1974. Computer methods
for sampling from Gamma, Beta, Poisson and Binomial distributions.
Computing 12: 223-246.*/

 float gammaMultiplication ( float x )
/*When alpha is an integer*/
     {
     int j;
     float p, RandomU, alphaInt;

     alphaInt = floor(x);
     j = 1;
     p = j;
     RandomU = rand()/((double)RAND_MAX+1);
     p = p*RandomU;
     while ( j != alphaInt ) {
         RandomU = rand()/((double)RAND_MAX+1);
         p = p*RandomU;
         j+=1;
     }
     return -( log(p) ) ;
     }

 float gammaRejection ( float x )
/*when alpha is real and <1*/
     {
     float RandomU, RandomU2, FuncB, FuncP, xGamma;

     RandomU = rand()/((double)RAND_MAX+1);
     FuncB = ( ( exp(1) + x)/exp(1) );
     FuncP = FuncB*RandomU;
     if (FuncP > 1 ) {
         xGamma = -log( ( FuncB-FuncP)/x );
         RandomU2 = rand()/((double)RAND_MAX+1);
     }
     else {
         xGamma = pow( FuncP, (1/x) );
         RandomU2 = rand()/((double)RAND_MAX+1);
     }
```

```
     while ( ( RandomU2 > exp (-xGamma) ) || ( RandomU2 > pow
(xGamma, (x-1) ) ) ) {
             RandomU = rand()/((double)RAND_MAX+1);
          FuncB = ( ( exp(1) + x)/exp(1) );
          FuncP = FuncB*RandomU;
     if (FuncP > 1 ) {
          xGamma = -log( ( FuncB-FuncP)/x );
          RandomU2 = rand()/((double)RAND_MAX+1);
     }
     else {
          xGamma = pow( FuncP, (1/x) );
          RandomU2 = rand()/((double)RAND_MAX+1);
     }
     }
     return xGamma;
     }


 float gammaTwoPart ( float x )
/*when alpha is real and >1*/
     {
     int j;
          float alphaInt, p, RandomU, RandomU2;
     float xGammaTwoPart1;
     float xGammaTwoPart2;
     float FuncB, FuncP;

     alphaInt = floor ( x );
     j = 1;
     p = j;
     RandomU = rand()/((double)RAND_MAX+1);
     p = p*RandomU;
     while ( j != alphaInt ) {
          RandomU = rand()/((double)RAND_MAX+1);
          p = p*RandomU;
          j+=1;
     }
     xGammaTwoPart1 = -log(p);

          RandomU = rand()/((double)RAND_MAX+1);
     FuncB = ( ( exp(1) + x)/exp(1) );
     FuncP = FuncB*RandomU;
     if (FuncP > 1 ) {
          xGammaTwoPart2 = -log( ( FuncB-FuncP)/x );
          RandomU2 = rand()/((double)RAND_MAX+1);
     }
     else {
          xGammaTwoPart2 = pow( FuncP, (1/x) );
          RandomU2 = rand()/((double)RAND_MAX+1);
     }

     while ( ( RandomU2 > ( exp (-xGammaTwoPart2) ) ) || ( RandomU2 >
pow (xGammaTwoPart2, (x-1) ) ) ) {
             RandomU = rand()/((double)RAND_MAX+1);
          FuncB = ( ( exp(1) + x)/exp(1) );
          FuncP = FuncB*RandomU;
          if (FuncP > 1 ) {
            xGammaTwoPart2 = -log( ( FuncB-FuncP)/x );
            RandomU2 = rand()/((double)RAND_MAX+1);
          }
          else {
```

```
            xGammaTwoPart2 = pow( FuncP, (1/x) );
            RandomU2 = rand()/((double)RAND_MAX+1);
          }
       }
       return xGammaTwoPart1 + xGammaTwoPart2;
     }

float normal ( float sd )
{

    float normalX;
    float u1, u2;

    u1 = rand()/((double)RAND_MAX+1);
    u2 = rand()/((double)RAND_MAX+1);
    normalX = sqrt(-2*log(u1))*sin(2*PI*u2);
    normalX = normalX*sd;
    return normalX;

}

float sumofsquares ( float a[], float Y, int x)
{
  float value=0, SumSquares=0;
  int i;

  for ( i = 0; i <= x-1; i ++) {
    value = pow( (a [ i ] - Y), 2 );
    SumSquares+=value;
  }
  return SumSquares;
}

float mean ( float a[], int x )
{
  float value=0;
  float Sum=0;
  float average=0;
  int i;

  for ( i = 0; i < x; i++ ) {
    value = a [ i ];
    Sum+=value;
  }
  average = Sum/(double)x;
  return average;
}
```

# Appendix III

## CoalFace Kylix/Delphi Source Code

"…tracing the ancestry of a gene backward in time and building up the family tree of the genes (at a particular locus) in a population sample back to the point at which they have a single common ancestor."

Kingman 2000

```
unit Unit1;

interface

uses
  SysUtils, Types, Classes, Variants, QMenus, QTypes, QGraphics, QControls,
QForms,
  QDialogs, QStdCtrls, QExtCtrls, Math, QComCtrls, QButtons, QFileCtrls;

type
  TfrmRandCoal = class(TForm)
    PageControl1: TPageControl;
    TabSheet1: TTabSheet;
    TabSheet2: TTabSheet;
    TabSheet3: TTabSheet;
    lblMutationRateExpl: TLabel;
    lblMutationRate: TLabel;
    lblPopulationSize: TLabel;
    lblSampleSize: TLabel;
    edtSampleSize: TEdit;
    edtPopulationSize: TEdit;
    edtMutationRate: TEdit;
    chkScatterMut: TCheckBox;
    chkSegSites: TCheckBox;
    lblSequenceLength: TLabel;
    edtSeqLength: TEdit;
    chkSeqInput: TCheckBox;
    chkShapeParameter: TCheckBox;
    chkInvarSites: TCheckBox;
    edtAlpha: TEdit;
    edtSeqFile: TEdit;
    edtI: TEdit;
    chkMultipleSims: TCheckBox;
    edtNumberSimulations: TEdit;
    dlgSaveFile: TSaveDialog;
    btnRunSimulation: TButton;
    lblDNAParams: TLabel;
    lblDNASeqParam: TLabel;
    Label6: TLabel;
    Image1: TImage;
    chkDrawMut: TCheckBox;
    edtFreqA: TEdit;
    edtFreqC: TEdit;
    edtFreqG: TEdit;
    lblFreqA: TLabel;
    lblFreqC: TLabel;
    lblFreqG: TLabel;
    TabSheet4: TTabSheet;
    lblFactorsN: TLabel;
    chkVarRepSuc: TCheckBox;
```

chkFluctuatingN: TCheckBox;
edtVarRepSuc: TEdit;
edtGrowthStart: TEdit;
lblGrowthStart: TLabel;
lblGrowthEnd: TLabel;
edtGrowthEnd: TEdit;
lblFlucN: TLabel;
edtChangeN: TEdit;
radJC69: TRadioButton;
radF81: TRadioButton;
radK2P: TRadioButton;
radHKY85: TRadioButton;
lblBaseFrequencies: TLabel;
lblTiTvRatio: TLabel;
edtTiTv: TEdit;
rdgOutput: TRadioGroup;
lblMsatParam: TLabel;
lblNoLoci: TLabel;
edtNumLoci: TEdit;
radInfiniteAlleles: TRadioButton;
radStepwise: TRadioButton;
Label1: TLabel;
edtMsatIn: TEdit;
chkMsats: TCheckBox;
radAllele: TRadioButton;
TabSheet5: TTabSheet;
lblGeneral: TLabel;
lblNexusTrees: TLabel;
lblTMRCA: TLabel;
lblSeqDataOut: TLabel;
edtSeqOut: TEdit;
lblMsatOut: TLabel;
edtTmrca: TEdit;
edtMSatData: TEdit;
edtNewickTrees: TEdit;
chkArlequin: TCheckBox;
lblLogFile: TLabel;
edtLogFile: TEdit;
ProgressBar1: TProgressBar;
Label2: TLabel;
Label3: TLabel;
Label4: TLabel;
Label5: TLabel;
Image3: TImage;
Label7: TLabel;
chkDiploid: TCheckBox;
chkSeqOut: TCheckBox;
chkMSatOut: TCheckBox;
lblMutDistrib: TLabel;
lblCoalDistrib: TLabel;

```
    edtMutDistrib: TEdit;
    edtCoalDistrib: TEdit;
    memSimNum: TMemo;
    chkSeqDiversity: TCheckBox;
    chkMSatDiversity: TCheckBox;
    lblOutDir: TLabel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    Label8: TLabel;
    Label9: TLabel;
    memDir: TMemo;
    BitBtn3: TBitBtn;
    DirectoryTreeView1: TDirectoryTreeView;
    edtMeanOffsp: TEdit;
    Label10: TLabel;

    procedure RunSimulation(Sender: TObject);
    //procedure Scale(Sender: TObject);

    procedure SaveGenealogy(Sender: TObject);
    procedure CreateForm(Sender: TObject);
    procedure AssignDirectory(Sender: TObject);
    procedure TreeFwd(Sender: TObject);
    procedure TreeBack(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmRandCoal : TfrmRandCoal;
  RandomkArray : array of integer;
  CountCoalesce : integer;
  CountCoalesce2 : integer;
  CountMutation : integer;
  NoMutations : integer;
  SortedRandArray : array of integer;
  CoalesceTime : integer;
  SampleSize : integer;
  TreeMatrix : array of array of integer;
  MutationMatrix : array of array of integer;
  BranchLengths : array of array of integer;
  CoalGenerations : array of array of integer;
  MutationScatter : array of array of integer;
  NewickArray : array of array of string;
  NewickArray2 : array of array of string;
  Data : array of array of char;
  DiploidData : array of array of char;
```

```
MData : array of array of integer;
DivInd : array of array of real;
NumberMutationArray : array of integer;
CoalArrayLength : integer;
N : integer;
NumberSimulations : integer;
outfile: textfile;
outfileN : Textfile;
outfileTMRCA : Textfile;
outfileCoalDis : Textfile;
outfileMuDis : Textfile;
GammaOut : textfile;
GammaOutPerSite : textfile;
ArlequinBatch : TextFile;
ArlequinProject : TextFile;
ScreenRatio : integer;
MuRatePerGen : extended;
MuRate : extended;
SeqLength : extended;
SeqLengthInt : integer;
NumberMutations : integer;
kSamples : integer;
MutLineages : array of array of integer;
MutLineages2 : array of array of integer;
pseudoNewick : string;
pseudoNewickArray : array of string;
sequence : array of char;
LineageClusters : array of array of string;
NucDiv, NucDivFin : real;
SegSites : integer;
TotalSegSitesFin : integer;
MuPerGene : extended;
FactorialResult, CombResult : extended;
xGamma : real;
NFactor : real;
StartN : integer;
NextN : integer;
NextNReal : real;
AddInd : real;
code : integer;
EndofRun : boolean;
locNumber, SizeLow, SizeHigh, repeatType : integer;
FileCounter : string;
ArlequinSampleSize : string;
valueChar : char;
NewickTreeFile : textfile;
Tmrca : array of integer;
TmrcaMSat : array of array of integer;
InfAlleles : array of integer;
InfAllelesMSat : array of array of integer;
```

```
  FinAlleles : array of integer;
  FinAllelesMSat : array of array of integer;
  GammaPerSiteAr : array of real;
  GammaGeneMuAr : array of real;
  AllelicDiversity : array of real;
  NucleotideDiversity : array of real;
  GeneDiversity : array of array of real;
  DirOut : string;
  TreeMatrixFile : textfile;
  MutationMatrixFile : textfile;
  TreeCounter : integer;
  SimDone : boolean;

const
  DevelopHeight = 1200;

procedure ChooseRandom(k, N : integer);
procedure SortRandom(Arrayk : array of integer; SampleSize : integer);
procedure MakeTree(Arrayk: array of integer; SampleSize: integer);
procedure Mutations(Samples: integer);
procedure NewickTree(Currentk, Originalk: integer);
procedure TrueNewickTree(Samples: integer);
procedure IdentifyLineageClusters(k : integer);
procedure MutateLineages(SeqLengthORLocus: integer);
procedure DiversityIndices(SeqLength, kSamples, SimNum: integer);
procedure MutationDistribution(Samples : integer);
procedure GeneMuRate(MuRatePerSitePerGen : extended; GeneLength: integer);
procedure MicrosatModel(Samples, PopSize, GStart, Gend, NSims : integer);
procedure GammaMutations(Samples,SeqLength, NumSims: integer;
alpha,MuRatePerGen: real);
procedure MicrosatMutations(Samples: integer; MicroRate : real);
procedure LogFile(kL,NL,StartNL,SeqLengthL,GrowthStartL,GrowthEndL,
NumberSimulationsL: integer; StartTimeL, EndTimeL : TDateTime;
SeqRateL,FreqAL,FreqCL,FreqGL,FreqTL : real);
procedure Outputs(NumberSims,NLoci,NumSamples: integer);
procedure DrawTree;

function JC69(CurrentBase: char): char;
function Kimura2Param(CurrentBase: char): char;
function Felsenstein81(CurrentBase: char): char;
function HKY85(CurrentBase: char): char;
function Factorial(Number: integer): extended;
function nChooseR (val1, val2 : integer) :extended;
function GammaMultiplication(a: real):real;
function GammaRejection(a: real): real;
function GammaTwoPart(a: real): real;
function MsatRandom(allele: integer): integer;
function MsatStepwise(allele: integer): integer;
```

```
implementation

{$R *.xfm}
{
procedure TfrmRandCoal.Scale(Sender: TObject);       //need to fix the resolution
problems, each button needs to be scaled accordingly

var
  ScreenHeight : integer;
  ScreenWidth : integer;

begin
  with Screen do
    ScreenHeight := Height;
    ScreenWidth := Width;
    ScreenRatio := DevelopHeight div ScreenHeight;
    ScaleBy(ScreenHeight,DevelopHeight);
    frmRandCoal.Height := ScreenHeight;
    frmRandCoal.Width := ScreenWidth;
end;
}
procedure ChooseRandom(k, N : integer);

//Chooses k Random numbers from N and assigns values to an array

var
  RandomK : integer;
  i : integer;

begin
  RandomkArray := nil;
  SetLength(RandomkArray, k);
  for i:=1 to k do begin
    Randomk := Random(N);
    RandomkArray[i-1] := RandomK;
  end;
end;

procedure SortRandom(Arrayk : array of integer; SampleSize : integer);

//Sorts the random values chosen in ChooseRandom

var
  sorted : boolean;
  l,i : integer;
  temp : integer;

begin
  repeat
```

```
    sorted := True;
    for l:=1 to SampleSize do begin
      if (Arrayk[l-1] > Arrayk[l]) then begin
        temp := Arrayk[l-1];
        Arrayk[l-1] := Arrayk[l];
        Arrayk[l] := temp;
        sorted:=False;
      end;
    end;
  until sorted;
  SortedRandArray := nil;
  setLength(SortedRandArray, SampleSize);
  for i := 1 to SampleSize do begin
    temp := Arrayk[i-1];
    SortedRandArray[i-1] := temp;
  end;
end;

procedure MakeTree(Arrayk: array of integer; SampleSize: integer);

//Identifies equal numbers in the k chosen from N, and assigns each identity
//event as a coalescent event. Also keeps track of the number of generations
//or the time to coalescence of each coalescent event. This procedure also
//writes the coalescent events with times and the descendents to a matrix called
//TreeMatrix.

var
  t : integer;
  TermA, TermB : integer;
  Diff : integer;
  CountCoalesceStart : integer;

begin
  CountCoalesceStart := CountCoalesce; //This code and the if loop ensures a max of
one coalescent
                                //...event per generation. You can allow polytomies by
removing it, but then it gets difficult to draw the tree and Newick format
    for t := 1 to SampleSize-1 do begin
      TermA := Arrayk[t-1];
      TermB := Arrayk[t];
      Diff := TermA - TermB;
      if (Diff = 0) and (CountCoalesce = CountCoalesceStart) then begin  //allow
polytomies by removing 2nd condition
        CountCoalesce := CountCoalesce+1;
        TreeMatrix[CountCoalesce-1, 1] := t-1;
        TreeMatrix[CountCoalesce-1, 2] := t;
        TreeMatrix[CountCoalesce-1, 0] := CoalesceTime;
      end;
    end;
end;
```

```
procedure GeneMuRate(MuRatePerSitePerGen : extended; GeneLength: integer);

begin
  MuPerGene := 1-Power((1-MuRatePerSitePerGen),GeneLength);
end;

function Factorial(Number: integer): extended;

begin
  FactorialResult := 1;
  If Number > 1 then begin
  repeat
    FactorialResult := FactorialResult*Number;
    Number := Number-1;
  until Number = 1;
  end;
end;

function nChooseR (val1, val2 : integer) :extended;

var
TermA, TermB, TermC : extended;

begin
  CombResult := 1;
  Factorial(val1);
  TermA := FactorialResult;
  Factorial(val2);
  TermB := FactorialResult;
  Factorial(val1-val2);
  TermC := FactorialResult;
  CombResult := TermA/((TermB)*(TermC));
end;

procedure Mutations(Samples: integer);

var
  RandomValue : extended;
  RandomLineage : integer;
  pMuPbPg : extended;
  pMuPbPgi : extended;
  CumPMuPbPgi : extended;
  i : integer;
  combin : extended;

begin
  RandomValue := Random;
  i := 1;
  pMuPbPg := 1-Power((1-MuPerGene), Samples);
```

```
  CumPMuPbPgi := 0;
  while ((PMuPbPg-CumPMuPbPgi) > RandomValue) do begin
    CountMutation := CountMutation+1;
    MutationMatrix[CountMutation-1, 0] := CoalesceTime;
    RandomLineage := Random(Samples);
    MutationMatrix[CountMutation-1, 1] := RandomLineage;
    nChooseR(Samples, i);
    Combin := CombResult;
    pMuPbPgi := (Power(MuPerGene,i)*Power((1-MuPerGene), Samples-i)*Combin);
    CumPMuPbPgi := CumPMuPbPgi+PMuPbPgi;
    i := i+1;
  end;
end;

function GammaMultiplication(a: real):real;
//the multiplication method for generating a gamma random variable when alpha
//is an integer. Ahrens & Dieter 1974 Computer methods for sampling from Gamma
//Beta, Poisson and Binomial distributions. Computing 12: 223-246

var
  j : integer;
  aInt : integer;
  RandomU : real;
  p : real;

begin
  aInt := Trunc(a);
  j := 1;
  p := j;
  RandomU := Random;
  p := p*RandomU;
  while (j <> aInt) do begin
    RandomU := Random;
    p := p*RandomU;
    j := j+1;
  end;
  xGamma := -ln(p);
end;

function GammaRejection(a: real): real;
//this function uses a rejection method to draw from a gamma distribution.
//Ahrens & Dieter 1974 Computer methods for sampling from Gamma
//Beta, Poisson and Binomial distributions. Computing 12: 223-246

var
  RandomU : real;
  RandomU2 : real;
  FuncB : real;
  FuncP : real;
```

```
begin
  repeat
    RandomU := random;
    FuncB := (exp(1) + a)/exp(1);
    FuncP := FuncB*RandomU;
    if FuncP > 1 then begin
      xGamma := -ln((FuncB-FuncP)/a);
      RandomU2 := random;
    end
    else begin
      xGamma := Power(FuncP, (1/a));
      RandomU2 := random;
    end;
  until (RandomU2 < exp(-xGamma)) or (RandomU2 < Power(xGamma, (a-1)));
end;

function GammaTwoPart(a: real): real;
//if alpha is real and > 1 then we use a two-part method to sample from the
//gamma distribution. this method breaks the real number into an integer part, and
//a real part and uses the methods above in combination.
//Ahrens & Dieter 1974 Computer methods for sampling from Gamma
//Beta, Poisson and Binomial distributions. Computing 12: 223-246

var
  aInt : integer;
  j : integer;
  p : real;
  RandomU : real;
  RandomU2 : real;
  xGammaTwoPart1 : real;
  xGammaTwoPart2 : real;
  FuncB, FuncP : real;

begin
  aInt := Trunc(a);
  j := 1;
  p := j;
  RandomU := Random;
  p := p*RandomU;
  while (j <> aInt) do begin
    RandomU := Random;
    p := p*RandomU;
    j := j+1;
  end;
  xGammaTwoPart1 := -ln(p);
  repeat
    RandomU := random;
    FuncB := (exp(1) + a)/exp(1);
    FuncP := FuncB*RandomU;
    if FuncP > 1 then begin
```

```
      xGammaTwoPart2 := -ln((FuncB-FuncP)/a);
      RandomU2 := random;
    end
    else begin
      xGammaTwoPart2 := Power(FuncP, (1/a));
      RandomU2 := random;
    end;
  until (RandomU2 < exp(-xGammaTwoPart2)) or (RandomU2 <
Power(xGammaTwoPart2, (a-1)));
  xGamma := xGammaTwoPart1+xGammaTwoPart2;
end;

procedure GammaMutations(Samples,SeqLength,NumSims : integer;
alpha,MuRatePerGen: real);

var
  RandomValue : extended;
  RandomLineage : integer;
  pMuPbPg : extended;
  pMuPbPgi : extended;
  CumPMuPbPgi : extended;
  i : integer;
  combin : extended;
  testAlpha : real;
  MuPerGenG : real;
  GammaGeneMu : real;
  StringG : string;

begin

  testAlpha := alpha-Trunc(alpha);
  if (testAlpha = 0) then  //i.e. is alpha an integer
    GammaMultiplication(alpha)
  else if (testAlpha <> 0) and (alpha > 0) and (alpha < 1) then //i.e. 0 < alpha < 1
    GammaRejection(alpha)
  else if (testAlpha <> 0) then
    GammaTwoPart(alpha); //i.e alpha is real and > 1

  MuPerGenG := (XGamma/alpha)*MuRatePerGen;
  GammaPerSiteAr[NumSims-1] := MuPerGenG;
  GammaGeneMu := 1-Power((1-MuPerGenG),SeqLength);
  GammaGeneMuAr[NumSims-1] := GammaGeneMu;

  RandomValue := Random;
  i := 1;
  pMuPbPg := 1-Power((1-MuPerGene), Samples);
  CumPMuPbPgi := 0;
  while ((PMuPbPg-CumPMuPbPgi) > RandomValue) do begin
    CountMutation := CountMutation+1;
    MutationMatrix[CountMutation-1, 0] := CoalesceTime;
```

```
    RandomLineage := Random(Samples);
    MutationMatrix[CountMutation-1, 1] := RandomLineage;
    nChooseR(Samples, i);
    Combin := CombResult;
    pMuPbPgi := (Power(MuPerGene,i)*Power((1-MuPerGene), Samples-i)*Combin);
    CumPMuPbPgi := CumPMuPbPgi+PMuPbPgi;
    i := i+1;
  end;
end;

procedure MicrosatMutations(Samples: integer; MicroRate : real);

var
  RandomValue : extended;
  RandomLineage : integer;
  pMuPbPg : extended;
  pMuPbPgi : extended;
  CumPMuPbPgi : extended;
  i : integer;
  combin : extended;

begin
  RandomValue := Random;
  i := 1;
  pMuPbPg := 1-Power((1-MicroRate), Samples);
  CumPMuPbPgi := 0;
  while ((PMuPbPg-CumPMuPbPgi) > RandomValue) do begin
    CountMutation := CountMutation+1;
    MutationMatrix[CountMutation-1, 0] := CoalesceTime;
    RandomLineage := Random(Samples);
    MutationMatrix[CountMutation-1, 1] := RandomLineage;
    nChooseR(Samples, i);
    Combin := CombResult;
    pMuPbPgi := (Power(MicroRate,i)*Power((1-MicroRate), Samples-i)*Combin);
    CumPMuPbPgi := CumPMuPbPgi+PMuPbPgi;
    i := i+1;
  end;
end;

procedure NewickTree(Currentk, Originalk: integer);

{Please note that the structure of this tree is not completely in Newick format. The
coalescent
events are correct but the branch lengths are not. The purpose of the branch lengths in
this
tree is to allow us to determine which samples cluster at which times such that we can
identify
the correct lineages to mutate when a mutation occurs. Later in the program, when the
TreeMatrix
is complete, I assemble a true Newick format tree}
```

```
var
  CoalEvents : integer;
  SampleA, SampleB : integer;
  NewickA, NewickB : string;
  BrLengthInt : integer;
  BrLength : string;
  i,j : integer;
  NewickCell : string;


begin
  CoalEvents := Originalk-Currentk;
  If (CoalEvents = 1) then begin
    for i:=0 to Originalk-1 do begin
      NewickArray[CoalEvents-1, i] := IntToStr(i);
    end;
  end;
  BrLengthInt := TreeMatrix[CoalEvents-1, 0];
  BrLength := IntToStr(BrLengthInt);
  SampleA := TreeMatrix[CoalEvents-1, 1];
  SampleB := TreeMatrix[CoalEvents-1, 2];
  NewickA := NewickArray[CoalEvents-1, SampleA];
  NewickB := NewickArray[CoalEvents-1, SampleA+1];
  j := 0;
  while (j < Originalk) do begin
    NewickArray[CoalEvents, j] := NewickArray[CoalEvents-1, j];;
    If (j = SampleA) then begin
      NewickCell := '('+NewickA+','+NewickB+'):'+BrLength;
      NewickArray[CoalEvents, j] := NewickCell;
    end;
    If (j >= SampleB) and (j <> Originalk-1) then begin
      NewickCell := NewickArray[CoalEvents-1, j+1];
      NewickArray[CoalEvents, j] := NewickCell;
    end;
    j := j+1;
  end;
end;

procedure TrueNewickTree(Samples: integer);

var
  i,j,l,m,n,o : integer;
  q,t,u,v,w,x : integer;
  SampleA, SampleB : integer;
  NewickA, NewickB : string;
  Branch1 : integer;
  Branch2A, Branch2B : integer;
  BranchDiffA, BranchDiffB : integer;
  TestChar, TestChar2 : char;
```

```
  ParenthesesA, ParenthesesB : boolean;
  BranchStr : string;
  NewickCell : string;
  BranchInt,BranchAdd : integer;
  BranchTest2 : integer;
  Count40_A, Count40_B : integer;
  Count40Test, Count41Test : integer;


begin
  NewickArray2 := nil;
  SetLength(NewickArray2, Samples, Samples);

  for i := 0 to Samples-1 do begin
    NewickArray2[0,i] := IntToStr(i);
  end;

  for j:= 0 to Samples-2 do begin
    SampleA := TreeMatrix[j, 1];
    SampleB := TreeMatrix[j, 2];
    NewickA := NewickArray2[j, SampleA];
    NewickB := NewickArray2[j, SampleA+1];
    Branch1 := TreeMatrix[j,0];

    l := 0;
    while (l < Samples) do begin
      NewickArray2[j+1, l] := NewickArray2[j, l];;
      If (l = SampleA) then begin

        //are there parentheses in NewickA or NewickB
        ParenthesesA := False;
        ParenthesesB := False;
        m := 1;
        repeat
          TestChar := NewickA[m];
          If (Ord(TestChar) = 40) then
            ParenthesesA := True;
          m := m+1;
        until (Ord(TestChar) = 40) or (m >= Length(NewickA));

        n := 1;
        repeat
          TestChar := NewickB[n];
          If (Ord(TestChar) = 40) then
            ParenthesesB := True;
          n := n+1;
        until (Ord(TestChar) = 40) or (n >= Length(NewickB));

        If (ParenthesesA = True) and (ParenthesesB = False) then begin
          BranchAdd := 0;
```

```
        q := Length(NewickA);
        repeat
          TestChar := NewickA[q];
          SetLength(BranchStr, 0);
          if (Ord(TestChar) = 58) then begin
            o := q+1;
            repeat
              TestChar := NewickA[o];
              BranchStr := BranchStr+TestChar;
              o := o+1;
            until (Ord(TestChar)=44) or (Ord(TestChar)=40) or (Ord(TestChar)=41);
            SetLength(BranchStr, Length(BranchStr)-1);
            BranchAdd := BranchAdd + StrToInt(BranchStr);
          end;
          q := q-1;
        until (Ord(TestChar) = 44);

        BranchDiffA := Branch1-(BranchAdd);
        NewickCell :=
'('+NewickA+':'+IntToStr(BranchDiffA)+','+NewickB+':'+IntToStr(Branch1)+')';
        NewickArray2[j+1, l] := NewickCell;
      end

      else If (ParenthesesA = False) and (ParenthesesB = True) then begin
        BranchAdd := 0;
        q := Length(NewickB);
        repeat
          TestChar := NewickB[q];
          SetLength(BranchStr, 0);
          if (Ord(TestChar) = 58) then begin
            o := q+1;
            repeat
              TestChar := NewickB[o];
              BranchStr := BranchStr+TestChar;
              o := o+1;
            until (Ord(TestChar)=44) or (Ord(TestChar)=40) or (Ord(TestChar)=41);
            SetLength(BranchStr, Length(BranchStr)-1);
            BranchAdd := BranchAdd + StrToInt(BranchStr);
          end;
          q := q-1;
        until (Ord(TestChar) = 44);

        BranchDiffB := Branch1-(BranchAdd);
        NewickCell :=
'('+NewickA+':'+IntToStr(Branch1)+','+NewickB+':'+IntToStr(BranchDiffB)+')';
        NewickArray2[j+1, l] := NewickCell;
      end

      else if (ParenthesesA = True) and (ParenthesesB = True) then begin
```

```
        BranchAdd := 0;

        q := Length(NewickA);
        repeat
          TestChar := NewickA[q];
          SetLength(BranchStr, 0);
          if (Ord(TestChar) = 58) then begin
            o := q+1;
            repeat
              TestChar := NewickA[o];
              BranchStr := BranchStr+TestChar;
              o := o+1;
            until (Ord(TestChar)=44) or (Ord(TestChar)=40) or (Ord(TestChar)=41);
            SetLength(BranchStr, Length(BranchStr)-1);
            BranchAdd := BranchAdd + StrToInt(BranchStr);
          end;
          q := q-1;
        until (Ord(TestChar) = 44);

        BranchDiffA := Branch1-(BranchAdd);

        BranchAdd := 0;
        q := Length(NewickB);
        repeat
          TestChar := NewickB[q];
          SetLength(BranchStr, 0);
          if (Ord(TestChar) = 58) then begin
            o := q+1;
            repeat
              TestChar := NewickB[o];
              BranchStr := BranchStr+TestChar;
              o := o+1;
            until (Ord(TestChar)=44) or (Ord(TestChar)=40) or (Ord(TestChar)=41);
            SetLength(BranchStr, Length(BranchStr)-1);
            BranchAdd := BranchAdd + StrToInt(BranchStr);
          end;
          q := q-1;
        until (Ord(TestChar) = 44);

        BranchDiffB := Branch1-(BranchAdd);

        NewickCell :=
'('+NewickA+':'+IntToStr(BranchDiffA)+','+NewickB+':'+IntToStr(BranchDiffB)+')';
        NewickArray2[j+1, l] := NewickCell;
      end

      else begin
        NewickCell :=
'('+NewickA+':'+IntToStr(Branch1)+','+NewickB+':'+IntToStr(Branch1)+')';
        NewickArray2[j+1, l] := NewickCell;
```

```
      end;
    end;

    If (l >= SampleB) and (l <> Samples-1) then begin
      NewickCell := NewickArray2[j, l+1];
      NewickArray2[j+1, l] := NewickCell;
    end;
    l := l+1;
  end;
 end;
end;

procedure IdentifyLineageClusters(k : integer);

var
 i,j,m,l : integer;
 clusterTime : integer;
 outfile : textfile;
 TextString : string;

begin
 LineageClusters := nil;
 SetLength(LineageClusters, k-1, k+1);
 for i:=0 to High(TreeMatrix) do begin
  clusterTime := TreeMatrix[i, 0];
  LineageClusters[i, 0] := IntToStr(clusterTime);
  for j:=1 to k-1 do begin
   LineageClusters[i,j] := NewickArray[i+1,j-1];
  end;
 end;
end;

procedure MutateLineages(SeqLengthORLocus : integer);

var
 MutationTime : integer;
 MutLineage, MutLineage2 : integer;
 ii,jj,nn,mm : integer;
 CurrentBase : char;
 RandomSite : integer;
 clusterTime : integer;
 clusterTimeStr : string;
 LineageString : string;
 CharN : char;
 CharNMinus : char;
 CharNPlus : char;
 result : char;
 resultm : integer;
 StringN : string;
 Count : integer;
```

```
  RandomAllele : real;
  RandomAlleleInt : integer;
  CurrentAllele : integer;

begin
  LineageString := 'd';
  StringN := 'k';
  MutLineage2 := 0;

  for ii:=CountMutation-1 downto 0 do begin     //Starting at the last mutation in the
genealogy...

    //the ff code identifies the time and lineage of the iith mutation. The iith lineage in
this case is one
    //of the (k-number Coalescent events) lineages in the current waiting time.
Therefore, we need to identify
    //the lineage clusters at each mutation time, and assign the cluster of lineages
mutated by the mutation
    //to a string LineageString.

    MutationTime := MutationMatrix[ii,0];
    MutLineage := MutationMatrix[ii,1];
    jj := 0;
    clusterTimeStr := LineageClusters[jj,0];
    clusterTime := StrToInt(clusterTimeStr);
    while (MutationTime > clusterTime) do begin
      jj := jj+1;
      clusterTimeStr := LineageClusters[jj,0];
      clusterTime := StrToInt(clusterTimeStr);
    end;

    If (jj = 0) then begin   //i.e. mutation occurs before coalescent events and therefore
MutLineage is the Sample
                          //number and only one sample should be mutated
      LineageString := IntToStr(MutLineage);
    end
    else begin
    LineageString := LineageClusters[jj-1, MutLineage+1];
    end;

    //now we find the current base/allele of the first sample in the lineage cluster to be
mutated

    nn := 1;
    CharN := LineageString[nn];
    Count := 0;   //since we only need the current base of the first lineage to input to the
mutation model
    while (Count < 1) and (nn <= Length(LineageString)) do begin
```

```
    if (Ord(CharN) <> 40) and (Ord(CharN) <> 41) and (Ord(CharN) <> 44) and
(Ord(CharN) <> 58) then begin //if char is not a ( ) , or : then it must be a lineage or
coalescent time
      CharNMinus := LineageString[nn-1];
      if (Ord(CharNMinus) <> 58) and (nn < Length(LineageString)) then begin
//char is not a coalescent time
       CharNPlus := LineageString[nn+1];   //char following is what?
       StringN := CharN;
       while (Ord(CharNPlus) <> 40) and (Ord(CharNPlus) <> 41) and
(Ord(CharNPlus) <> 44) and (Ord(CharNPlus) <> 58) and (nn <
Length(LineageString)) do begin
         StringN := StringN+CharNPlus;    //if ff char is not (),: then it is a number that
should be collated with the sample eg. ( 3 4 , = 34
         nn := nn+1;
         CharNPlus := LineageString[nn+1];
       end;
       MutLineage2 := StrToInt(StringN);
       Count := Count+1;
      end
      else if (Ord(CharNMinus) = 58) then begin     //if preceding char is a : then the
number is a coalescent time
       CharNPlus := LineageString[nn+1];
       while (Ord(CharNPlus) <> 44) and (Ord(CharNPlus) <> 40) and
(Ord(CharNPlus) <> 41) do begin //or part of a coalescent time
         nn := nn+1;
         CharNPlus := LineageString[nn+1];
       end;
      end;
    end;
    nn := nn+1;
    CharN := LineageString[nn];
  end;

  //now we determine the mutation that occurs from current base to ? at a random site
  //under the selected mutation model for sequences or microsatellite model

  with frmRandCoal do begin
   if chkMsats.Checked then begin
    CurrentAllele := MData[MutLineage2, SeqLengthORLocus];
    if radAllele.Checked then
      resultm := MsatRandom(CurrentAllele);
    if radStepwise.Checked then
      resultm := MSatStepwise(CurrentAllele);
   end
   else begin
    RandomSite := Random(SeqLengthORLocus);  //between 0 and 999 which is
fine since Data is an array with index 0>999
    CurrentBase := Data[RandomSite, MutLineage2];
    If radJC69.checked then
      result := JC69(CurrentBase);
```

```
      If radK2P.checked then
        result := Kimura2Param(CurrentBase);
      If radF81.checked then
        result := Felsenstein81(CurrentBase);
      If radHKY85.checked then
        result := HKY85(CurrentBase);
    end;
  end;


  //now we enforce the above mutations on the samples in the current lineage cluster

  mm := 1;
  repeat

    CharN := LineageString[mm];
    if (Ord(CharN) <> 40) and (Ord(CharN) <> 41) and (Ord(CharN) <> 44) and
(Ord(CharN) <> 58) then begin
      CharNMinus := LineageString[mm-1];

      if (Ord(CharNMinus) <> 58) and (mm <> Length(LineageString)) then begin
        CharNPlus := LineageString[mm+1];
        StringN := CharN;
        while (Ord(CharNPlus) <> 40) and (Ord(CharNPlus) <> 41) and
(Ord(CharNPlus) <> 44) and (Ord(CharNPlus) <> 58) and (mm <>
Length(LineageString)) do begin
          StringN := StringN+CharNPlus;
          mm := mm+1;
          CharNPlus := LineageString[mm+1];
        end;
        MutLineage2 := StrToInt(StringN);
        with frmRandCoal do begin
          if chkMsats.Checked then begin
            MData[MutLineage2, SeqLengthORLocus] := resultm;
          end
          else begin
            Data[RandomSite, MutLineage2] := result;
          end;
        end;
      end;

      if (Ord(CharNMinus) = 58) then begin
        CharNPlus := LineageString[mm+1];
        while (Ord(CharNPlus) <> 44) and (Ord(CharNPlus) <> 40) and
(Ord(CharNPlus) <> 41) and (mm < Length(LineageString)) do begin
          mm := mm+1;
          CharNPlus := LineageString[mm+1];
        end;
      end;
```

```
      end;
      mm := mm+1;

    until (mm >= Length(LineageString));

  end;
end;



function JC69(CurrentBase: char): char;

var
  RandomMutation : integer;

begin
  case Ord(CurrentBase) of
    65, 97 : begin     //a, A
               RandomMutation := Random(3);
               case RandomMutation of
                 0 : begin
                       result := 'c';
                     end;
                 1 : begin
                       result := 'g';
                     end;
                 2 : begin
                       result := 't';
                     end;
               end;
             end;
    67, 99 : begin     //c, C
               RandomMutation := Random(3);
               case RandomMutation of
                 0 : begin
                       result := 'a';
                     end;
                 1 : begin
                       result := 'g';
                     end;
                 2 : begin
                       result := 't';
                     end;
               end;
             end;
    71,103 : begin     //g, G
               RandomMutation := Random(3);
               case RandomMutation of
                 0 : begin
                       result := 'a';
```

```
              end;
            1 : begin
                result := 'c';
              end;
            2 : begin
                result := 't';
              end;
          end;
        end;
  84,116 : begin     //t, T
          RandomMutation := Random(3);
          case RandomMutation of
            0 : begin
                result := 'a';
              end;
            1 : begin
                result := 'c';
              end;
            2 : begin
                result := 'g';
              end;
          end;
        end;
 end;
end;

function Kimura2Param(CurrentBase: char): char;

var
 RandomMutation : integer;
 TiTvRatio : real;
 code : integer;
 RandomAdd : real;
 RAI : integer;

begin
 with frmRandCoal do begin
  Val(edtTiTv.Text, TiTvRatio, code);
 end;
 RandomAdd := TiTvRatio*100;   //TiTv ratio must be to 2 decimal places or less
 RAI := Trunc(RandomAdd);

 case Ord(CurrentBase) of
  65, 97 : begin     //a, A
          RandomMutation := Random(200+RAI);
          if (RandomMutation >=0) and (RandomMutation < RAI) then
            result := 'g'     //Ti
          else if (RandomMutation >= RAI) and (RandomMutation < RAI+100) then
            result := 'c'     //Tv
```

```
            else if (RandomMutation >= RAI+100) and (RandomMutation < RAI+200)
then
              result := 't';     //Tv
            end;
    67, 99 : begin     //c, C
            RandomMutation := Random(200+RAI);
            if (RandomMutation >=0) and (RandomMutation < RAI) then
              result := 't'     //Ti
            else if (RandomMutation >= RAI) and (RandomMutation < RAI+100) then
              result := 'a'     //Tv
            else if (RandomMutation >= RAI+100) and (RandomMutation < RAI+200)
then
              result := 'g';     //Tv
            end;
    71,103 : begin     //g, G
            RandomMutation := Random(200+RAI);
            if (RandomMutation >=0) and (RandomMutation < RAI) then
              result := 'a'     //Ti
            else if (RandomMutation >= RAI) and (RandomMutation < RAI+100) then
              result := 'c'     //Tv
            else if (RandomMutation >= RAI+100) and (RandomMutation < RAI+200)
then
              result := 't';     //Tv
            end;
    84,116 : begin     //t, T
            RandomMutation := Random(200+RAI);
            if (RandomMutation >=0) and (RandomMutation < RAI) then
              result := 'c'     //Ti
            else if (RandomMutation >= RAI) and (RandomMutation < RAI+100) then
              result := 'a'     //Tv
            else if (RandomMutation >= RAI+100) and (RandomMutation < RAI+200)
then
              result := 'g';     //Tv
            end;
  end;
end;


function Felsenstein81(CurrentBase: char): char;

var
  FreqA, FreqC, FreqG, FreqT : real;
  code : integer;
  RandomMutation : integer;
  RandomFrom : real;
  RFI : integer;

begin
  with frmRandCoal do begin
    Val(edtFreqA.Text, FreqA, code);
```

```
  Val(edtFreqC.Text, FreqC, code);
  Val(edtFreqG.Text, FreqG, code);
 end;

 FreqT := 1-(FreqA+FreqC+FreqG);

 case Ord(CurrentBase) of
  65, 97 : begin     //a, A
            RandomFrom := (FreqC+FreqG+FreqT)*100;
            RFI := Trunc(RandomFrom);
            RandomMutation := Random(RFI);
            if (RandomMutation >=0) and (RandomMutation < FreqC*100) then
              result := 'c'
            else if (RandomMutation >= FreqC*100) and (RandomMutation <
(FreqC+FreqG)*100) then
              result := 'g'
            else if (RandomMutation >= (FreqC+FreqG)*100) and (RandomMutation <
(FreqC+FreqG+FreqT)*100) then
              result := 't';
           end;
  67, 99 : begin     //c, C
            RandomFrom := (FreqA+FreqG+FreqT)*100;
            RFI := Trunc(RandomFrom);
            RandomMutation := Random(RFI);
            if (RandomMutation >=0) and (RandomMutation < FreqA*100) then
              result := 'a'
            else if (RandomMutation >= FreqA*100) and (RandomMutation <
(FreqA+FreqG)*100) then
              result := 'g'
            else if (RandomMutation >= (FreqA+FreqG)*100) and (RandomMutation <
(FreqA+FreqG+FreqT)*100) then
              result := 't';
           end;
  71,103 : begin     //g, G
            RandomFrom := (FreqA+FreqC+FreqT)*100;
            RFI := Trunc(RandomFrom);
            RandomMutation := Random(RFI);
            if (RandomMutation >=0) and (RandomMutation < FreqA*100) then
              result := 'a'
            else if (RandomMutation >= FreqA*100) and (RandomMutation <
(FreqA+FreqC)*100) then
              result := 'c'
            else if (RandomMutation >= (FreqA+FreqC)*100) and (RandomMutation <
(FreqA+FreqC+FreqT)*100) then
              result := 't';
           end;
  84,116 : begin     //t, T
            RandomFrom := (FreqA+FreqC+FreqG)*100;
            RFI := Trunc(RandomFrom);
            RandomMutation := Random(RFI);
```

```
            if (RandomMutation >=0) and (RandomMutation < FreqA*100) then
              result := 'a'
            else if (RandomMutation >= FreqA*100) and (RandomMutation <
(FreqA+FreqC)*100) then
              result := 'c'
            else if (RandomMutation >= (FreqA+FreqC)*100) and (RandomMutation <
(FreqA+FreqC+FreqG)*100) then
              result := 'g';
          end;
  end;

end;

function HKY85(CurrentBase: char): char;

var
  FreqA, FreqC, FreqG, FreqT : real;
  code : integer;
  RandomMutation : integer;
  TiTvRatio : real;
  RandomFrom : real;
  RFI : integer;


begin
  with frmRandCoal do begin
    Val(edtFreqA.Text, FreqA, code);
    Val(edtFreqC.Text, FreqC, code);
    Val(edtFreqG.Text, FreqG, code);
    Val(edtTiTv.Text, TiTvRatio, code);
  end;
  FreqT := 1-(FreqA+FreqC+FreqG);

  case Ord(CurrentBase) of
    65, 97 : begin     //a, A
            RandomFrom := (TiTvRatio*100*freqG)+(100*freqC)+(100*freqT);
            RFI := Trunc(RandomFrom);
            RandomMutation := Random(RFI);
            if (RandomMutation >=0) and (RandomMutation <
(TiTvRatio*100*freqG)) then
              result := 'g'     //Ti
            else if (RandomMutation >= TiTvRatio*100*freqG) and (RandomMutation
< ((TiTvRatio*100*freqG)+(100*freqC))) then
              result := 'c'     //Tv
            else if (RandomMutation >= ((TiTvRatio*100*freqG)+(100*freqC))) and
(RandomMutation < (TiTvRatio*100*freqG)+(100*freqC)+(100*freqT)) then
              result := 't';     //Tv
          end;
    67, 99 : begin     //c, C
            RandomFrom := (TiTvRatio*100*freqT)+(100*freqA)+(100*freqG);
```

```
            RFI := Trunc(RandomFrom);
            RandomMutation := Random(RFI);
            if (RandomMutation >=0) and (RandomMutation < (TiTvRatio*100*freqT))
then
              result := 't'     //Ti
            else if (RandomMutation >= TiTvRatio*100*freqT) and (RandomMutation
< ((TiTvRatio*100*freqT)+(100*freqA))) then
              result := 'a'     //Tv
            else if (RandomMutation >= ((TiTvRatio*100*freqT)+(100*freqA))) and
(RandomMutation < (TiTvRatio*100*freqT)+(100*freqA)+(100*freqG)) then
              result := 'g';     //Tv
          end;
   71,103 : begin     //g, G
           RandomFrom := (TiTvRatio*100*freqA)+(100*freqC)+(100*freqT);
            RFI := Trunc(RandomFrom);
            RandomMutation := Random(RFI);
            if (RandomMutation >=0) and (RandomMutation <
(TiTvRatio*100*freqA)) then
              result := 'a'     //Ti
            else if (RandomMutation >= TiTvRatio*100*freqA) and (RandomMutation
< ((TiTvRatio*100*freqA)+(100*freqC))) then
              result := 'c'     //Tv
            else if (RandomMutation >= ((TiTvRatio*100*freqA)+(100*freqC))) and
(RandomMutation < (TiTvRatio*100*freqA)+(100*freqC)+(100*freqT)) then
              result := 't';     //Tv
          end;
   84,116 : begin     //t, T
            RandomFrom := (TiTvRatio*100*freqC)+(100*freqA)+(100*freqG);
            RFI := Trunc(RandomFrom);
            RandomMutation := Random(RFI);
            if (RandomMutation >=0) and (RandomMutation < (TiTvRatio*100*freqC))
then
              result := 'c'     //Ti
            else if (RandomMutation >= TiTvRatio*100*freqC) and (RandomMutation
< ((TiTvRatio*100*freqC)+(100*freqA))) then
              result := 'a'     //Tv
            else if (RandomMutation >= ((TiTvRatio*100*freqC)+(100*freqA))) and
(RandomMutation < (TiTvRatio*100*freqC)+(100*freqA)+(100*freqG)) then
              result := 'g';     //Tv
          end;
  end;
end;

function MsatRandom(allele: integer): integer;   //choose a random allele within the
size range

var
  SizeDiff : integer;
  NewAllele : integer;
```

```
begin
  SizeDiff := SizeHigh-SizeLow;
  repeat
    NewAllele := Trunc(Random(SizeDiff));
  until (NewAllele mod repeatType = 0);
  result := NewAllele+SizeLow;
end;

function MsatStepwise(allele: integer): integer;  //choose an allele one step size
smaller or larger than the current

var
  NewAllele : integer;

begin
  if (Random > 0.5) then
    NewAllele := allele+repeatType
  else
    NewAllele := allele-repeatType;
  result := NewAllele;
end;

procedure MutationDistribution(Samples : integer);

var
  i,j,k,l,m: integer;
  MutTime : integer;
  CoalTime : integer;
  DiffTime : integer;
  TreeLength : array of array of integer;
  BranchLengths : array of array of integer;
  TotalTreeLength : integer;
  MuTreeLength : integer;
  outfile : textfile;
  value : integer;
  result : string;
  BranchLength : integer;
  MutDistrib : array of array of integer;


begin
  TotalTreeLength := 0;
  TreeLength := nil;
  BranchLengths := nil;
  SetLength(TreeLength, Samples-1, 2);
  SetLength(BranchLengths, Samples-1, 2);
  for i:=0 to High(TreeMatrix) do begin
    TreeLength[i,0] := TreeMatrix[i,0];
    if (i=0) then begin
      TreeLength[i,1] := TreeMatrix[i,0]*(Samples);
```

```
    end
    else begin
      TreeLength[i,1] := (TreeMatrix[i,0]-TreeMatrix[i-1,0])*(Samples-
i)+TreeLength[i-1,1];
      BranchLengths[i,1] := (TreeMatrix[i,0]-TreeMatrix[i-1,0])*(Samples-i);
    end;
  end;

  //check code for tree length calculations

  Append(outfileCoalDis);
  Writeln(outfileCoalDis, 'Time of Coalescence, Total genealogy length');
  for j := 0 to High(TreeLength) do begin
    value := TreeLength[j,0];
    Str(value, result);
    Write(outfileCoalDis, result);
    Write(outfileCoalDis, ',');
    value := TreeLength[j,1];
    Str(value, result);
    Write(outfileCoalDis, result);
    Writeln(outfileCoalDis);
  end;
  CloseFile(outfileCoalDis);

  MutDistrib := nil;
  setLength(MutDistrib, CountMutation, 2);

  for k:=0 to CountMutation-1 do begin
    MuTreeLength := 0;
    MutTime := MutationMatrix[k,0];
    l := 0;
    CoalTime := TreeMatrix[l,0];

    //mutation occurs before any coalescent events
    if (MutTime < CoalTime) then begin
      DiffTime := CoalTime-MutTime;
      MuTreeLength := DiffTime*Samples;
    end

    //mutation occurs after coalescent events and treelength is calculated up to
    //when coalescent time > mutime
    else begin
      while (CoalTime < MutTime) do begin
        BranchLength := BranchLengths[l,1];
        MuTreeLength := MuTreeLength + BranchLength;
        l := l+1;
        CoalTime := TreeMatrix[l,0];
      end;
```

```
    //when mutation occurs between coalescent events the difference is
added*numberSamples
      if (MutTime < CoalTime) then begin
        DiffTime := MutTime-TreeMatrix[l-1,0];
        MuTreeLength := MuTreeLength+(DiffTime*(Samples-l));
      end;
    end;

    MutDistrib[k,0] := MutTime;
    MutDistrib[k,1] := MuTreeLength;

  end;

  Append(outfileMuDis);
  Writeln(outfileMuDis, 'Time of mutation, Total tree length');
  for m:=0 to High(MutDistrib) do begin

    value := MutDistrib[m,0];
    str(value, result);
    Write(outfileMuDis, result);
    Write(outfileMuDis, ',');

    value := MutDistrib[m,1];
    str(value, result);
    Write(outfileMuDis, result);

    Writeln(outfileMuDis);
  end;
  CloseFile(outfileMuDis);
end;

procedure DiversityIndices(SeqLength, kSamples, SimNum : integer);

var
  i,j,l : integer;
  m,n,o : integer;
  Base1, Base2 : char;
  SegSitesFin : integer;
  Sequence1, Sequence2 : integer;
  Base1U, Base2U : char;
  CountDifferences : integer;
  PairwiseDist : real;
  PairwiseDistArray : array of real;
  CountZero : integer;
  Base1Str, Base2Str : string;
  SumPairwise : real;

begin
  SegSites := 0;
  with frmRandCoal do begin
```

```
if chkSegSites.checked then begin
  SegSites := CountMutation;
end;
if chkScatterMut.checked then begin
  for m:=0 to SeqLength-1 do begin
    n := 0;
    SegSitesFin := 0;
    while (SegSitesFin <> 1) and (n <= kSamples-2) do begin
      Base1 := Data[m,0];
      Base2 := Data[m, n+1];
      if (Ord(Base1) <> Ord(Base2)) then begin
        SegSitesFin := SegSitesFin+1;
      end;
      n := n+1;
    end;
    TotalSegSitesFin := TotalSegSitesFin+SegSitesFin;
  end;
end;

if chkSeqDiversity.checked then begin
  l := 0;
  CountZero := 0;
  PairwiseDistArray := nil;
  SetLength(PairwiseDistArray, ((kSamples*(kSamples-1)) div 2));
  for i := 1 to kSamples-1 do begin
    Sequence1 := i;
    Sequence2 := i;
    repeat
      CountDifferences := 0;
      Sequence2 := Sequence2+1;
      for j:=1 to SeqLength do begin
        Base1 := Data[j-1,Sequence1-1];
        Base2 := Data[j-1,Sequence2-1];
        Base1Str := UpperCase(Base1);
        Base2Str := UpperCase(Base2);
        Base1 := Base1Str[1];
        Base2 := Base2Str[1];
        if (Ord(Base1) <> Ord(Base2)) then
          CountDifferences := CountDifferences+1;
      end;
      PairwiseDist := CountDifferences/SeqLength;
      if (PairwiseDist > 0) then begin
        PairwiseDistArray[l] := PairwiseDist;
        l := l+1;
      end;
      if (PairwiseDist = 0) then begin
        CountZero := CountZero+1;
      end;
    until (Sequence2 = kSamples);
  end;
```

```
      AllelicDiversity[SimNum-1] := (((kSamples*(kSamples-1))/2-
CountZero))/((kSamples*(kSamples-1))/2);
      o := 0;
      repeat
        PairwiseDist := PairwiseDistArray[o];
        SumPairWise := SumPairWise+PairwiseDist;
        o := o+1;
      until (PairwiseDist = 0);
      NucleotideDiversity[SimNum-1] := (2/(kSamples*(kSamples-1)))*SumPairWise;
    end;
  end;
end;

procedure MicrosatModel(Samples, PopSize, GStart, GEnd, NSims : integer);

var
  i,j,q,s,t,a,b : integer;
  ri,rj,rk,rl,rm,rn,ro,rp,rq,rt,rr,rs : integer;
  gd : integer;
  ai, aj, ak : integer;
  MDataTemp, MDataTemp2 : array of array of integer;
  SortedMData : array of array of integer;
  AlleleFrequency : array of real;
  tempInt, tempInt2 : integer;
  CountAlleles : integer;
  NumberAlleles : array of integer;
  value2, value3 : integer;
  MsatInput : Textfile;
  NumLoci : integer;
  MSatMuRate : real;
  StartAllele1, StartAllele2 : integer;
  RangeSize : integer;
  RandomAllele : real;
  IntRandomAllele : integer;
  MDataOut : TextFile;
  CurrentAllele : integer;
  AlleleInt : integer;
  AlleleStr : string;
  MAlleles, MAlleles2 : TextFile;
  value : integer;
  result : string;
  sorted : boolean;
  StrValue : string;
  RandomChosenM : array of integer;
  ts, tt : integer;
  RandomMSat : real;
  RandomMSatInt : integer;
  AlreadyUsed : boolean;
  tempRandom : integer;
  individ : integer;
```

```
  Allele1 : integer;
  StringName : string;
  AlleleCount : integer;
  AlleleFreq : real;
  SumSquares : real;
  tmf,tmf2 : integer;
  mmf,mmf2 : integer;
  NumberSoFar : integer;

begin
  SampleSize := Samples;

  with frmRandCoal do begin
    AssignFile(MsatInput, DirOut + '/' + edtMsatIn.Text);
    Val(edtNumLoci.Text, NumLoci, code);
    AssignFile(NewickTreeFile, DirOut + '/' + edtNewickTrees.Text);
    Rewrite(NewickTreeFile);
    if chkMSatOut.checked then begin
      AssignFile(MDataOut, DirOut + '/' + edtMSatData.Text);
      Rewrite(MDataOut);
    end;
  end;

  Tmrca := nil;
  InfAllelesMSat := nil;
  FinAllelesMSat := nil;
  GeneDiversity := nil;
  SetLength(TmrcaMsat, NSims, NumLoci);
  SetLength(InfAllelesMsat, NSims, NumLoci);
  SetLength(FinAllelesMSat, NSims, NumLoci);
  SetLength(GeneDiversity, NSims, NumLoci);

  AssignFile(TreeMatrixFile, DirOut + '/' + 'TreeMatFile.txt');
  Rewrite(TreeMatrixFile);
  AssignFile(MutationMatrixFile, DirOut + '/' + 'MutMatFile.txt');
  Rewrite(MutationMatrixFile);
  NumberMutationArray := nil;
  SetLength(NumberMutationArray, NumberSimulations*NumLoci);
  NumberSoFar := 0;

  for j:=1 to NSims do begin
    with frmRandCoal do begin
      memSimNum.Clear;
      memSimNum.Lines.Add(IntToStr(j));
      frmRandCoal.Repaint;
    end;
    MData := nil;
    SetLength(MData, Samples, NumLoci);
    Reset(MSatInput);
    for i := 0 to NumLoci-1 do begin
```

```
N := PopSize;
Read(MsatInput, locNumber);
Read(MsatInput, SizeLow);
Read(MsatInput, SizeHigh);
Read(MsatInput, repeatType);
Read(MsatInput, MSatMuRate);
Readln(MsatInput);  //skips to next line and next locus
RangeSize := SizeHigh-SizeLow;

repeat
  RandomAllele := Random(RangeSize);
  IntRandomAllele := Trunc(RandomAllele);
until (IntRandomAllele mod repeatType = 0);
  StartAllele1 := IntRandomAllele+SizeLow;

//Create identical starting msat alleles (those before mutation) in an array, with
//length/rows = # samples and width/columns = loci.

for q := 0 to (Samples)-1 do begin
  MData[q, i] := StartAllele1;
end;

CountCoalesce := 0;
CountCoalesce2 := 0;
CountMutation := 0;
CoalesceTime := 0;
CoalArrayLength := Samples-1;
SampleSize := Samples;
TreeMatrix := nil;
MutationMatrix := nil;
MutationScatter := nil;
MutLineages := nil;
MutLineages2 := nil;
NewickArray := nil;
SetLength(TreeMatrix, SampleSize-1, 3);
SetLength(MutationMatrix, 1000000, 2);
SetLength(MutationScatter, CoalArrayLength, CoalArrayLength);
SetLength(MutLineages, CoalArrayLength, Samples+1);
SetLength(MutLineages2, CoalArrayLength, Samples+1);
SetLength(NewickArray, Samples+1, Samples+1);

with frmRandCoal do begin
  ProgressBar1.FillColor := clBlue;
  ProgressBar1.Max := (Samples-1)*(NumLoci)*NumberSimulations;
end;

N := 2*N;

repeat //This repeat loop moves through one generation at a time, asking whether
there is a coalescent or mutation event.
```

```
        with frmRandCoal do begin
          If (chkFluctuatingN.checked) and (CoalesceTime > GStart) and (CoalesceTime
< GEnd) then begin
            StartN := N;
            Val(edtChangeN.Text, NFactor, code);
            AddInd := ((NFactor/100)*StartN);
            NextNReal := StartN+AddInd;
            NextN := Trunc(NextNReal);
            N := NextN;
          end;
        end;
        CoalesceTime := CoalesceTime+1;
        EndofRun := True;
        ChooseRandom(SampleSize,N);
        SortRandom(RandomkArray, SampleSize);
        MakeTree(SortedRandArray, SampleSize);
        MicrosatMutations(SampleSize, MsatMuRate);
        If CountCoalesce > CountCoalesce2 then begin
          SampleSize := Samples-CountCoalesce;
          NewickTree(SampleSize, Samples);
          with frmRandCoal do
            ProgressBar1.StepBy(1);
        end;
        CountCoalesce2 := CountCoalesce;
        if CountCoalesce < Samples-1 then
          EndofRun:=False;
      until EndofRun;

      TmrcaMSat[j-1,i] := CoalesceTime;

      //mutate the MData array with the relevant mutation model, before Mutation
Matrix
      //is reset on the simulation for the next locus
      //in the case of an infinite alleles model, write the number of alleles to a file

      IdentifyLineageClusters(Samples);
      MutateLineages(i);
      MutationDistribution(Samples);

      InfAllelesMSat[j-1,i]:=CountMutation+1;


      Append(NewickTreeFile);
      TrueNewickTree(Samples);
      StrValue := NewickArray2[Samples-1,0];
      Writeln(NewickTreeFile, StrValue + ';');
      CloseFile(NewickTreeFile);

      Append(TreeMatrixFile);
      Writeln(TreeMatrixFile, '#' + IntToStr(NumberSoFar+i+1));
```

```
    for tmf2 := 0 to Samples-2 do begin
     for tmf := 0 to 2 do begin
       value := TreeMatrix[tmf2, tmf];
       Write(TreeMatrixFile, IntToStr(value) + ' ');
     end;
     Writeln(TreeMatrixFile);
    end;
    CloseFile(TreeMatrixFile);

    Append(MutationMatrixFile);
    Writeln(MutationMatrixFile, '#' + IntToStr(NumberSoFar+i+1));
    for mmf := 0 to CountMutation-1 do begin
     for mmf2 := 0 to 1 do begin
       value := MutationMatrix[mmf,mmf2];
       Write(MutationMatrixFile, IntToStr(value) + ' ');
     end;
     Writeln(MutationMatrixFile);
    end;
    CloseFile(MutationMatrixFile);
    NumberMutationArray[NumberSoFar+i] := CountMutation;

   end; //end of number of loci loop

   NumberSoFar := NumberSoFar + (i);

   with frmRandCoal do begin
    if radAllele.checked or radStepwise.Checked and chkMSatOut.checked then
begin

     RandomChosenM := nil;
     SetLength(RandomChosenM, Samples);
     ts := 0;
     repeat
      RandomMSat := Random(Samples+1);
      RandomMSatInt := Trunc(RandomMSat);
      AlreadyUsed := False;
      tt := 0;
      repeat
       tempRandom := RandomChosenM[tt];
       if (tempRandom = RandomMSatInt) then
         AlreadyUsed := True;
       tt := tt+1;
      until (AlreadyUsed = True) or (tt = ts+1);
      if (AlreadyUsed = False) then begin
       RandomChosenM[ts] := RandomMSatInt;
       ts := ts+1;
      end;
     until (ts = Samples);

     Individ := 1;
```

```
      Append(MDataOut);
      for s:= 0 to Samples-1 do begin
        Allele1 := RandomChosenM[s];
        if (Odd(s) = False) and (s <> 0) then begin
          Individ := Individ+1;
        end;
        if (Odd(s) = False) or (s = 0) then begin
          StringName := IntToStr(Individ) + ' ';
          Write(MDataOut, StringName:10);
        end
        else begin
          Write(MDataOut, '          ');
        end;

        Write(MDataOut, ' ');
        for t:=1 to NumLoci do begin
          AlleleInt := MData[Allele1-1,t-1];
          Str(AlleleInt, AlleleStr);
          Write(MDataOut, AlleleStr);
          Write(MDataOut, ' ');
        end;
        Writeln(MDataOut);
      end;
      Writeln(MDataOut);
      CloseFile(MDataOut);

      //

    end;
  end;

  with frmRandCoal do begin
    If chkArlequin.Checked then begin
      Append(ArlequinBatch);
      Individ := 1;
      FileCounter := DirOut + '/Arlequin/sim_' + IntToStr(j) + '.arp';
      Writeln(ArlequinBatch, FileCounter);
      CloseFile(ArlequinBatch);
      AssignFile(ArlequinProject, FileCounter);
      Rewrite(ArlequinProject);
      Writeln(ArlequinProject, '[Profile]');
      Writeln(ArlequinProject, '   Title="Microsatellite data generated using coalescent
simulations"');
      Writeln(ArlequinProject, '   NbSamples=1');
      Writeln(ArlequinProject, '   GenotypicData=1');
      Writeln(ArlequinProject, '   GameticPhase=0');
      Writeln(ArlequinProject, '   DataType=MICROSAT');
      Writeln(ArlequinProject, '   LocusSeparator=WHITESPACE');
      Writeln(ArlequinProject, '[Data]');
      Writeln(ArlequinProject, ' [[Samples]]');
```

```
    Writeln(ArlequinProject, '   SampleName="Panmictic population"');
    ArlequinSampleSize := IntToStr(Samples div 2);
    Writeln(ArlequinProject, '   SampleSize=' + ArlequinSampleSize);
    Writeln(ArlequinProject, '   SampleData= {');

    for ai := 0 to Samples-1 do begin
      Allele1 := RandomChosenM[ai];
      if (Odd(ai) = False) and (ai <> 0) then begin
        Individ := Individ+1;
      end;
      if (Odd(ai) = False) or (ai = 0) then begin
        StringName := IntToStr(Individ) + ' ';
        Write(ArlequinProject, StringName:10);
        Write(ArlequinProject, ' 1 ');
      end
      else begin
        Write(ArlequinProject, '            ');
      end;
      for aj:=1 to NumLoci do begin
        AlleleInt := MData[Allele1-1,aj-1];
        Str(AlleleInt, AlleleStr);
        Write(ArlequinProject, AlleleStr);
        Write(ArlequinProject, ' ');
      end;
      Writeln(ArlequinProject);
    end;
    Writeln(ArlequinProject, '}');
    CloseFile(ArlequinProject);
  end;
end;

//be careful here. the following code changes the array MData, such that we
//can count alleles. I cannot copy the array to a new independent array since
//the copy function will only copy single dimension arrays. i could copy each
//column and susequently each allele in each locus to a new array, in a loop that
//counts the loci*2, but its much easier to simply change the MData array, and then
//not to use it again...ie. trash it.

with frmRandCoal do begin
  if radAllele.checked or radStepwise.Checked then begin
    SortedMData := nil;
    setLength(SortedMData, Samples, NumLoci);

    for ri := 1 to NumLoci do begin
      repeat
        sorted := True;
        for rj:=1 to (Samples)-1 do begin
          if (MData[rj-1, ri-1] > MData[rj, ri-1]) then begin
            tempInt := MData[rj-1, ri-1];
            MData[rj-1, ri-1] := MData[rj, ri-1];
```

```
          MData[rj,ri-1] := tempInt;
          sorted:=False;
        end;
      end;
    until sorted;

    for rl := 1 to Samples do begin
      tempInt := MData[rl-1,ri-1];
      SortedMData[rl-1,ri-1] := tempInt;
    end;

  end;

  NumberAlleles := nil;
  SetLength(NumberAlleles, NumLoci);
  AlleleFrequency := nil;
  SetLength(AlleleFrequency, Samples);
  for rm :=1 to NumLoci do begin
    rs := 0;
    CountAlleles := 1;
    AlleleCount := 1;
    for rn := 1 to (Samples-2) do begin
      tempInt := SortedMData[rn-1, rm-1];
      tempInt2 := SortedMData[rn, rm-1];
      if (tempInt = tempInt2) then begin
        AlleleCount := AlleleCount + 1;
      end;
      if (tempInt <> tempInt2) or (rn = Samples-1) then begin
        AlleleFrequency[rs] := AlleleCount/Samples;
        CountAlleles := CountAlleles+1;
        AlleleCount := 1;
        rs := rs+1;
      end;

    end;
    NumberAlleles[rm-1] := CountAlleles;
    FinAllelesMSat[j-1,rm-1] := CountAlleles;

    gd := 0;
    SumSquares := 0;
    AlleleFreq := AlleleFrequency[gd];
    while (AlleleFreq <> 0) do begin
      SumSquares := SumSquares + Power(AlleleFreq,2);
      gd := gd+1;
      AlleleFreq := AlleleFrequency[gd];
    end;
    GeneDiversity[j-1,rm-1] := (Samples/(Samples-1))*(1-SumSquares);
  end;
 end;
end;
```

```
  end; // end of number of simulations loop
  CloseFile(MSatInput);
  Outputs(NumberSimulations,NumLoci,Samples);
  TreeCounter := NumberSimulations*NumLoci;
  DrawTree;
end;

procedure DrawTree;

var
  originX : integer;
  originY : integer;
  radius : real;
  rad : integer;
  TimePlus, PointInterval, TimeCoalEvent, DrawTo : real;
  TimePlusAbs : real;
  TimePlusAbsInt : integer;
  DrawToInt : integer;
  i,j,l, m, o, p, q, r, s, a, b, c, d, e, f, g, h, k, t, u, w, x, y, z: integer;
  aa, ab, ac, ad, ae, af, ag, ah, ai, aj, ak, al, am, an, ao, ap, mi : integer;
  CoalCount : integer;
  Counter : integer;
  endTime, endTime2 : integer;
  value : integer;
  result, result2 : string;
  TimeGenerations : real;
  Time2N : real;
  PI, PlacePoint : integer;
  NumberSamples : integer;
  XSamples : array of integer;
  XSamples2 : array of integer;
  CoalescentTimes : array of integer;
  Descendent : integer;
  BranchID : integer;
  XMove, XMoveTwo : integer;
  TimeSingle : integer;
  TimeSingleA, TimeSingleB : real;
  NumberCoalGen : integer;
  FirstDescendent : integer;
  Time, NewTime : integer;
  TimeInt, TimePlusInt : integer;
  TimeLast : integer;
  TreeWidth : integer;
  TimeCoalEventInt : integer;
  BranchLength : integer;
  CoalTime : integer;
  CoalTimeInt : integer;
  CoalTimeCan : real;
  StartX, StartY : integer;
  EndX, EndY : integer;
```

```
  NextCoalescent : integer;
  NodeID : array of integer;
  NumberNodes : integer;
  NumberNodesInCoalescent : integer;
  NodePlaceX : integer;
  NumberGenCoals : integer;
  GenerationA, GenerationB : integer;
  next : integer;
  sum : integer;
  tester : integer;
  GenA : integer;
  TreeMatrix2 : array of integer;
  difference : integer;
  countLength : integer;
  NumberEvents : integer;
  TimeNowInt, Xmove2, YMove2, YMove : integer;
  Descendent1 : integer;
  TimeNow : real;
  PILength : integer;
  end3 : integer;
  TimePlus2 : integer;
  TimeLast2 : integer;
  TimePlusInt2 : integer;
  Temp : integer;
  DescendentPrev : integer;
  VerticalLimit : real;
  HorizontalLimit : real;
  StepSize : real;
  NumberCoalEvents : integer;
  MutLineages : array of array of integer;
  RowDescendent : integer;
  TestOne, TestTwo, TestThree, TestA, TestB, TestC, TestD : integer;
  MutationGen : integer;
  MutationGenAbs : integer;
  MutationGenReal : real;
  MutationLineage : integer;
  TimePrev : integer;


begin
  with frmRandCoal do begin
    Image1.Picture := nil;
    VerticalLimit := 490;
    HorizontalLimit := 500;

    //Draw dashed lines at each coalescent time, and assign the position to a matrix
called
    //CoalescentTimes.
    endTime := CoalArrayLength;
    CoalescentTimes := nil;
```

```
  SetLength(CoalescentTimes, endTime);
  StepSize := VerticalLimit/(N*3);
  for i:=1 to endTime do begin
   TimeSingle := TreeMatrix[i-1, 0];
   TimeLast := TreeMatrix[endTime-1, 0];
   TimeGenerations := TimeSingle;
   //Time2N := (TimeSingle/(2*N));
   TimeCoalEvent := VerticalLimit-(TimeSingle*StepSize);
   TimeCoalEventInt := Trunc(TimeCoalEvent);
   CoalescentTimes[i-1] := TimeCoalEventInt;
   with Image1,Canvas do begin
    value := Trunc(TimeCoalEvent);
    Pen.Style := psDash;
    Pen.Color := clBlack;
    Pen.Width := 1;
    Brush.Color := clWhite;
    MoveTo(10, value);
    //LineTo(1400, value);   //Draws dotted lines across tree at each coalescent event
    Font.Color := clBlue;
    Font.Size := 10;
    str(TimeGenerations:10:0, result);
    //str(Time2N:10:0, result2);
    TextOut(510, value-Font.Size, result)
   end;
  end;

 //Draw the tree from the top down, the top being the extant genes. First, we
determine
 //the number of coalescent events in each generation. Thereafter, the coalescent
events are drawn for each generation.
 //The matrix CoalGenerations assembles a matrix of each generation in which there
is one or more coalescent events
 //and the first individual (of each pair) involved in a coalescent event. Column 0
provides a very useful
 //value; i.e. the number of coalescent events at each generation - only important if
we change the code back to allowing
 //multiple coalescent events and polytomies - though drawing trees and keeping
track of mutations becomes difficult
 //when the assumption of one coalescent per generation is relaxed

  CoalGenerations := nil;
  SetLength(CoalGenerations, CoalArrayLength, CoalArrayLength);

  j := 1;
  r := 1;
  while j < CoalArrayLength+1 do begin
   q := 4;
   NumberGenCoals := 1;
   GenerationA := (TreeMatrix[j-1,0]);
   CoalGenerations[r-1, 2] := TreeMatrix[j-1, 1];
```

```
  CoalGenerations[r-1, 1] := GenerationA;
  for p:=j+1 to CoalArrayLength do begin
    GenerationB := TreeMatrix[p-1, 0];
    difference := (GenerationA-GenerationB);
    If difference = 0 then begin
      NumberGenCoals := NumberGenCoals+1;
      CoalGenerations[r-1, q-1] := TreeMatrix[p-1, 1];
      q := q+1;
    end;
  end;
  j:=j+NumberGenCoals;
  CoalGenerations[r-1, 0] := NumberGenCoals;
  r:=r+1;
end;

NumberCoalGen := r-1;
endTime2 := NumberCoalGen-1;
TimeNow := 490;
TimeNowInt := Trunc(TimeNow);
NumberSamples := CoalArrayLength+1;
PointInterval := (HorizontalLimit/(NumberSamples));
PI := Trunc(PointInterval);
TreeWidth := (NumberSamples-1)*PI;
PlacePoint := 20+PI;

//Get your starting points and write them to a matrix called XSamples
XSamples := nil;
SetLength(XSamples, NumberSamples);
for c:=1 to NumberSamples do begin
  with Image1,Canvas do begin
    Pen.Style := psSolid;
    Radius := 3.0;
    rad := Trunc(Radius);
    Brush.Style := bsSolid;
    Brush.Color := clBlue;
    XSamples[c-1] := PlacePoint;
    PlacePoint := PlacePoint+PI;
  end;
end;

for a:=1 to NumberCoalGen do begin
  If (a = 1) then begin
    TimePrev := 0;
  end
  else begin
    TimePrev := CoalGenerations[a-2, 1];
  end;

  TimeLast := CoalGenerations[endTime2, 1];
  TimePlus := CoalGenerations[a-1, 1];
```

```
TimePlusAbsInt := Trunc(TimePlus);
TimePlus := VerticalLimit-(TimePlus*StepSize);
TimePlusInt := Trunc(TimePlus);
NumberEvents := CoalGenerations[a-1, 0];
XSamples2 := nil;
SetLength(XSamples2, NumberSamples);

for t:=1 to NumberSamples do begin
  XSamples2[t-1] := XSamples[t-1];
end;

for m:=1 to NumberSamples do begin
  PlacePoint := XSamples[m-1];
  with Image1,Canvas do begin
    Pen.Style := psSolid;
    OriginX := PlacePoint;
    OriginY := TimeNowInt;
    MoveTo(OriginX, OriginY);
    LineTo(OriginX, TimePlusInt);
  end;
end;

  //this tree drawing code can handle up to four samples coalescing into 1 in a single
event.
  //to be able to handle more you need to keep checking the subsequent descendent
against the first!
  //you can probably do this with a controlled forward loop...but its not neccessary
since I have now
  //forced only one coalescent event per generation!

  s:=2;
  u := 0;
  x := 0;

  while u < NumberSamples do begin
    Descendent1 := CoalGenerations[a-1, s];
    if Descendent1 = (u) then begin
      with Image1,Canvas do begin
        XMove := XSamples[Descendent1];
        MoveTo(XMove, TimePlusInt);
        XMoveTwo := XMove + ((XSamples[Descendent1+1]-XMove) div 2);
        DrawToInt := XSamples[Descendent1+1];
        LineTo(DrawToInt, TimePlusInt);
        XSamples2[Descendent1-(s-2)] := XMoveTwo;
        s:= s+1;
        u := u+2;
        DescendentPrev := Descendent1;
        Descendent1 := CoalGenerations[a-1, s];
        if Descendent1 = DescendentPrev+1 then begin
          XMove := XSamples[Descendent1+1];
```

```
            MoveTo(XMove, TimePlusInt);
            XMoveTwo := XMove - ((XMove - XSamples[DescendentPrev]) div 2);
            DrawToInt := XSamples[DescendentPrev];
            LineTo(DrawToInt, TimePlusInt);
            XSamples2[DescendentPrev+1-(s-2)] := XMoveTwo;
            s := s+1;
            u := u+1;
            DescendentPrev := Descendent1;
            Descendent1 := CoalGenerations[a-1, s];
            if Descendent1 = DescendentPrev+1 then begin
              XMove := XSamples[Descendent1+1];
              MoveTo(XMove, TimePlusInt);
              XMoveTwo := XMove - ((XMove - XSamples[DescendentPrev]) div 2);
              DrawToInt := XSamples[DescendentPrev];
              LineTo(DrawToInt, TimePlusInt);
              XSamples2[DescendentPrev+1-(s-2)] := XMoveTwo;
              s := s+1;
              u := u+1;
            end;
          end;
        end;
      end
      else begin
        XSamples2[(u)-(s-2)] := XSamples[u];
        u := u+1;
      end;
    end;

    if chkDrawMut.checked then begin
      if CountMutation > 0 then begin
        for mi := 0 to CountMutation-1 do begin
          MutationGenAbs := MutationMatrix[mi, 0];
          if (MutationGenAbs <= TimePlusAbsInt) and (MutationGenAbs > TimePrev)
then begin //i.e the mutation has occured b4 next coalescent event
            MutationLineage := MutationMatrix[mi, 1];
            XMove := XSamples[MutationLineage];
            MutationGenReal := VerticalLimit-(MutationGenAbs*StepSize);
            MutationGen := Trunc(MutationGenReal);
            with Image1,Canvas do begin
              MoveTo(XMove-4, MutationGen);
              Pen.Style := psSolid;
              Pen.Color := clRed;
              Pen.Width := 3;
              LineTo(XMove+4, MutationGen);
              MoveTo(5, MutationGen);
              Font.Color := clRed;
              Font.Size := 10;
              str(MutationGenAbs, result);
              TextOut(5, MutationGen-Font.Size, result);
              Pen.Width := 1;
```

```
            Pen.Color := clBlack;
           end;
          end;
         end;
        end;
       end;

       for w :=1 to NumberSamples do begin
        XSamples[w-1] := XSamples2[w-1];
       end;

       TimeNowInt := TimePlusInt;
       NumberSamples := NumberSamples-(NumberEvents);
     end;
     with Image1,Canvas do begin
       Font.Color := clBlack;
       Font.Size := 12;
       TextOut(250, 510, 'TMRCA = '+IntToStr(TimeLast));
     end;
   end;
end;

procedure LogFile(kL,NL,StartNL,SeqLengthL,GrowthStartL,GrowthEndL,
NumberSimulationsL: integer; StartTimeL, EndTimeL : TDateTime;
SeqRateL,FreqAL,FreqCL,FreqGL,FreqTL : real);

var
  logfile : TextFile;
  MSatIn : TextFile;
  strVal : string;
  model : string;
  titvReal : real;
  titv : string[4];
  invarstr : string;
  invarReal : real;
  shapestr : string;
  shapeReal : real;
  strA, strC, strG, strT : string;
  strTReal : real;
  strAReal : real;
  strCReal : real;
  strGReal : real;
  code : integer;
  valueReal : real;
  Result : string;
  i : integer;
  NumLociL : integer;
  locNumberInt, SizeLowInt, SizeHighInt, repeatTypeInt : integer;
  MSatMuRateReal : real;
  MSatMuRateStr : string;
```

```
  vkL : real;
  VkLStr : string;
  NChangeReal : real;
  NChangeStr : string;



begin
  with frmRandCoal do begin
    AssignFile(logfile, DirOut + '/' + edtLogFile.Text);
    Rewrite(logfile);
    Writeln(logfile, 'Coalescent Simulations generated with RandoCoal0.2b: LogFile');
    Writeln(logfile,
'************************************************************');
    Writeln(logfile);
    Writeln(logfile, DateToStr(Date));

    Writeln(logfile, 'Started at: ' + TimeToStr(StartTimeL));
    Writeln(logfile, 'Finished at: ' + TimeToStr(EndTimeL));
    Writeln(logfile);
    Writeln(logfile, 'Coalescent Parameters');
    Writeln(logfile, '**********');
    Writeln(logfile);
    if (chkFluctuatingN.Checked) then begin
      Writeln(logfile, 'Starting population Size: ' + IntToStr(StartNL));
      Writeln(logfile, 'Final population Size: ' + IntToStr(NL));
    end
    else begin
      Writeln(logfile, 'Population size: ' + IntToStr(NL));
    end;

    Writeln(logfile, 'Number of samples: ' + IntToStr(kL));
    Writeln(logfile, 'Number of simulations: ' + IntToStr(NumberSimulationsL));


    Writeln(logfile);
    Writeln(logfile, 'Mutation Parameters');
    Writeln(logfile, '**********');
    Writeln(logfile);

    if (chkMsats.Checked) then begin
      Writeln(logfile, 'Datatype: Microsatellites');
      AssignFile(MsatIn, DirOut + '/' + edtMsatIn.Text);
      Reset(MsatIn);
      Val(edtNumLoci.Text, NumLociL, code);
      Writeln(logfile, 'Number of loci: ' + IntToStr(NumLociL));
      if (radInfiniteAlleles.Checked) then
        Writeln(logfile, 'Mutation model: infinite allele')
      else if (radStepwise.Checked) then
        Writeln(logfile, 'Mutation model: Stepwise')
```

```
    else if (radAllele.Checked) then
      Writeln(logfile, 'Mutation model: Random allele');
    Writeln(logfile);
    Writeln(logfile, 'Locus #   Size Range   Repeat Type      Mutation Rate');
    Writeln(logfile, '*******   **********   ***********      *************');

    for i := 0 to NumLociL-1 do begin
      Read(MsatIn, locNumberInt);
      Read(MsatIn, SizeLowInt);
      Read(MsatIn, SizeHighInt);
      Read(MsatIn, repeatTypeInt);
      Read(MsatIn, MSatMuRateReal);
      Readln(MsatIn);  //skips to next line and next locus
      Str(MsatMuRateReal, MSatMuRateStr);
      Write(logfile, IntToStr(locNumberInt):3, IntToStr(SizeLowInt):10,
IntToStr(SizeHighInt):5, IntToStr(repeatTypeInt):8);
      Write(logfile, '          ' + MSatMuRateStr);
      Writeln(logfile);
    end;
    CloseFile(MSatIn);

    Writeln(logfile, 'Output Files');
    Writeln(logfile, '*************');
    Writeln(logfile);
    Writeln(logfile, 'Tmrca and diversity indices: ' + edtTmrca.Text);
    Writeln(logfile, 'Newick tree output file: ' + edtNewickTrees.Text);
    if chkMSatOut.Checked then
      Writeln(logfile, 'Microsatellite profiles generated: ' + edtMSatData.Text);
    Writeln(logfile, 'Distribution of coalescence times with genealogy length: ' +
edtCoalDistrib.Text);
    Writeln(logfile, 'Distribution of mutation times with genealogy length: ' +
edtMutDistrib.Text);

  end
  else begin
    Writeln(logfile, 'Datatype: DNA sequence');
    Str(SeqRateL, strVal);

    Writeln(logfile, 'Mutation rate: ' + strVal);
    Writeln(logfile, 'Sequence Length: ' + IntToStr(SeqLengthL));

    if (chkSeqInput.checked) then begin
      Writeln(logfile, 'Root sequence provided in ' + edtSeqFile.Text);
    end
    else begin
      Writeln(logfile, 'Root sequence generated at random');
    end;

    If (radJC69.Checked) then begin
      model := 'JC69';
```

```
  Writeln(logfile, 'Substitution model: ' + model);
  If (chkInvarSites.checked) then begin
    Val(edtI.Text, invarReal, code);
    Str(invarReal:0:3, invarstr);
    Writeln(logfile, 'Proportion of invariable sites: ' + invarstr);
  end;
  If (chkShapeParameter.checked) then begin
    Val(edtAlpha.Text, shapeReal, code);
    Str(shapeReal:0:3, shapestr);
    Writeln(logfile, 'Among-site rate variation shape parameter: ' + shapestr);
  end;
end;

if (radF81.Checked) then begin
  model := 'F81';
  Writeln(logfile, 'Substitution model: ' + model);
  Write(logfile, 'A:C:G:T = ');
  Str(FreqAL:0:2, strA);
  Str(FreqCL:0:2, strC);
  Str(FreqGL:0:2, strG);
  Str(FreqTL:0:2, strT);
  Write(logfile, strA + ':' + strC + ':' + strG + ':' + strT);

  Writeln(logfile);
  If (chkInvarSites.checked) then begin
    Val(edtI.Text, invarReal, code);
    Str(invarReal:0:3, invarstr);
    Writeln(logfile, 'Proportion of invariable sites: ' + invarstr);
  end;
  If (chkShapeParameter.checked) then begin
    Val(edtAlpha.Text, shapeReal, code);
    Str(shapeReal:0:3, shapestr);
    Writeln(logfile, 'Among-site rate variation shape parameter: ' + shapestr);
  end;
end;

if (radK2P.Checked) then begin
  model := 'Kimura 2-parameter';
  Writeln(logfile, 'Substitution model: ' + model);
  Val(edtTiTv.Text, titvReal, code);
  Str(titvReal:0:2, titv);
  Writeln(logfile, 'TiTv ratio: ' + titv);
  If (chkInvarSites.checked) then begin
    Val(edtI.Text, invarReal, code);
    Str(invarReal:0:3, invarstr);
    Writeln(logfile, 'Proportion of invariable sites: ' + invarstr);
  end;
  If (chkShapeParameter.checked) then begin
    Val(edtAlpha.Text, shapeReal, code);
    Str(shapeReal:0:3, shapestr);
```

```
        Writeln(logfile, 'Among-site rate variation shape parameter: ' + shapestr);
      end;
    end;

    if (radHKY85.Checked) then begin
      model := 'HKY85';
      Writeln(logfile, 'Substitution model: ' + model);
      Val(edtTiTv.Text, titvReal, code);
      Str(titvReal:0:2, titv);
      Writeln(logfile, 'TiTv ratio: ' + titv);
      Write(logfile, 'A:C:G:T = ');
      Str(FreqAL:0:2, strA);
      Str(FreqCL:0:2, strC);
      Str(FreqGL:0:2, strG);
      Str(FreqTL:0:2, strT);
      Write(logfile, strA + ':' + strC + ':' + strG + ':' + strT);
      Writeln(logfile);
      If (chkInvarSites.checked) then begin
        Val(edtI.Text, invarReal, code);
        Str(invarReal:0:3, invarstr);
        Writeln(logfile, 'Proportion of invariable sites: ' + invarstr);
      end;
      If (chkShapeParameter.checked) then begin
        Val(edtAlpha.Text, shapeReal, code);
        Str(shapeReal:0:3, shapestr);
        Writeln(logfile, 'Among-site rate variation shape parameter: ' + shapestr);
      end;
    end;
    Writeln(logfile);
    Writeln(logfile, 'Output Files');
    Writeln(logfile, '************');
    Writeln(logfile);
    Writeln(logfile, 'Tmrca and diversity indices: ' + edtTmrca.Text);
    Writeln(logfile, 'Newick tree output file: ' + edtNewickTrees.Text);
    if chkSeqOut.Checked then
      Writeln(logfile, 'Sequence data output file: ' + edtSeqOut.Text);
    Writeln(logfile, 'Distribution of coalescence times with genealogy length: ' +
edtCoalDistrib.Text);
    Writeln(logfile, 'Distribution of mutation times with genealogy length: ' +
edtMutDistrib.Text);


  end;

  if (chkVarRepSuc.Checked) or (chkFluctuatingN.Checked) then begin
    Writeln(logfile);
    Writeln(logfile, 'Demographics');
    Writeln(logfile, '************');
    Writeln(logfile);
    if (chkVarRepSuc.Checked) then begin
```

```
        Val(edtVarRepSuc.Text, VkL, code);
        Str(Vkl:0:3, VklStr);
        Writeln(logfile, 'Variance in reproductive success: ' + VkLStr);
      end;

      if (chkFluctuatingN.Checked) then begin
        Writeln(logfile, 'Population growth starts at generation: ' +
IntToStr(GrowthStartL));
        Writeln(logfile, 'Population growth ends at generation: ' +
IntToStr(GrowthEndL));
        Val(edtChangeN.Text, NChangeReal, code);
        Str(NChangeReal:0:2, NChangeStr);
        Writeln(logfile, 'Population size changes by ' + NChangeStr + '% each
generation');
      end;
    end;
    CloseFile(logfile);
  end;
end;

procedure Outputs(NumberSims, NLoci, NumSamples: integer);

var
  outfileAll : textfile;
  outfileAllM : textfile;
  Intvalue : integer;
  realValue : real;
  realString : string;
  i,j : integer;
  outfileSeq : textFile;
  StringReal : string;


begin
  with frmRandCoal do begin
    if chkMsats.Checked then begin
      AssignFile(outfileAllM, DirOut + '\' + edtTmrca.Text);
      Rewrite(outfileAllM);
      Write(outfileAllM, 'Simulation #, ');
      for j := 1 to NLoci do begin
        Write(outfileAllM, 'Tmrca:Loc ' + IntToStr(j) + ', ');
        Write(outfileAllM, 'Inf a:Loc ' + IntToStr(j) + ', ');
        Write(outfileAllM, 'Fin a:Loc ' + IntToStr(j) + ', ');
        if chkMSatDiversity.Checked then begin
          Write(outfileAllM, 'Allelic Diversity:Loc ' + IntToStr(j) + ', ');
          Write(outfileAllM, 'Gene Diversity:Loc ' + IntToStr(j) + ', ');
        end;
      end;
      Writeln(outfileAllM);
```

```
    for i := 1 to NumberSims do begin
      Write(outfileAllM, IntToStr(i) + ', ');
      for j := 1 to NLoci do begin
        Intvalue := TmrcaMSat[i-1,j-1];
        Write(outfileAllM, IntToStr(Intvalue) + ', ');
        Intvalue := InfAllelesMSat[i-1,j-1];
        Write(outfileAllM, IntToStr(Intvalue) + ', ');
        Intvalue := FinAllelesMSat[i-1,j-1];
        Write(outfileAllM, IntToStr(Intvalue) + ', ');
        if chkMsatDiversity.Checked then begin
          Intvalue := FinAllelesMsat[i-1,j-1];
          Realvalue := Intvalue/(NumSamples);
          Str(RealValue:6:3, StringReal);
          Write(outfileAllM, StringReal + ', ');
          Realvalue := GeneDiversity[i-1,j-1];
          Str(Realvalue:6:3, StringReal);
          Write(outfileAllM, StringReal + ', ');
        end;
      end;
      Writeln(outfileAllM);
    end;
    CloseFile(outfileAllM);
  end
  else begin
    AssignFile(outfileAll, DirOut + '\' + edtTmrca.Text);
    Rewrite(outfileAll);
    if chkSeqDiversity.Checked then
      Writeln(outfileAll, 'Simulation#, Tmrca, Infinite s, Finite s, Allelic Div,
Nucleotide Div')
    else
      Writeln(outfileAll, 'Simulation #, Tmrca, Infinite s, Finite s');
    for i := 1 to NumberSims do begin
      Write(outfileAll, IntToStr(i) + ', ');
      Intvalue := Tmrca[i-1];
      Write(outfileAll, IntToStr(Intvalue) + ', ');
      Intvalue := InfAlleles[i-1];
      Write(outfileAll, IntToStr(Intvalue) + ', ');
      Intvalue := FinAlleles[i-1];
      Write(outfileAll, IntToStr(Intvalue));
      if chkSeqDiversity.Checked then begin
        Realvalue := AllelicDiversity[i-1];
        str(RealValue:4:5, StringReal);
        Write(outfileAll, ', ' + StringReal + ', ');
        Realvalue := NucleotideDiversity[i-1];
        str(RealValue:4:5, StringReal);
        Write(outfileAll, StringReal);
      end;
      Writeln(outfileAll);
    end;
    CloseFile(outfileAll);
```

```
    end;
  end;
end;


procedure TfrmRandCoal.RunSimulation(Sender: TObject);

var
  k, code, j, l : integer;
  tmf, tmf2, mmf, mmf2 : integer;
  ai, aj : integer;
  m, p : integer;
  o, s : integer;
  value : integer;
  StrValue : string;
  result : string;
  //result1, result2 : string;
  //value1, value2 : integer;
  i,h,q,r,v,w,x,y : integer;
  rs : integer;
  //LastTime : integer;
  //Sequence : array of char;
  SeqLength : integer;
  RandomChar : integer;
  base : char;
  infile : TextFile;
  outfile, outfileSeq : TextFile;
  outfileSeqLength : TextFile;
  t, u : integer;
  seed : integer;
  resultString : string;
  ab, ac : integer;
  GrowthStart : integer;
  GrowthEnd : integer;
  Vk : real;
  NReal : real;
  PropInVar : real;
  SeqLengthReal : real;
  FreqA, FreqC, FreqG, FreqT : real;
  alpha : real;
  Nst : integer;
  StringShape : string;
  StartTime, EndTime : TDateTime;
  NBeforeGrowth : integer;
  RandomSequence : real;
  RandomSequenceInt : integer;
  RandomChosen : array of integer;
  AlreadyUsed : boolean;
  tempRandom : integer;
  rt,rc : integer;
```

```
  Individ : integer;
  RandomInd : integer;
  StringName : string;
  mk : integer;

begin
  StartTime := Time;
  NumberSimulations := 1;
  Randomize;
  Val(edtSampleSize.Text, k, code);
  Val(edtPopulationSize.Text, N, code);
  NBeforeGrowth := N;
  GrowthStart := 0;
  GrowthEnd := 0;

  with ProgressBar1 do begin
    Position := 0;
  end;

  AssignFile(outfileCoalDis, DirOut + '\' + edtCoalDistrib.Text);
  Rewrite(outfileCoalDis);

  AssignFile(outfileMuDis, DirOut + '\' + edtMutDistrib.Text);
  Rewrite(outfileMuDis);

  If (k > 0) and (N > 0) then begin
    If chkFluctuatingN.Checked then begin
      Val(edtGrowthStart.Text, GrowthStart, code);
      Val(edtGrowthEnd.Text, GrowthEnd, code);
    end;

    If chkMultipleSims.Checked then begin
      Val(edtNumberSimulations.Text, NumberSimulations, code);
    end;

    If chkVarRepSuc.checked then begin
      Val(edtVarRepSuc.Text, Vk, code);
      Val(edtMeanOffsp.Text, Mk, code);
      if chkDiploid.checked then begin
        NReal := ((N*Mk)-1)/((Mk-1+(Vk/Mk)));
        N := Trunc(NReal);
      end
      else begin
        NReal := (((N*Mk)-1))/((Mk-1+(Vk/Mk)))/2;
        N := Trunc(NReal);
      end;
    end;

    If chkScatterMut.Checked and chkSeqOut.checked then begin
      AssignFile(outfileSeq, DirOut + '\' + edtSeqOut.Text);
```

```
    Rewrite(outfileSeq);
  end;

  If chkArlequin.Checked and (chkSeqOut.Checked or chkMSatOut.Checked) then
begin
    CreateDir(DirOut + '\' + 'Arlequin');
    AssignFile(ArlequinBatch, DirOut + '\' + 'ArlequinBatch.arp');
    Rewrite(ArlequinBatch);
  end;

  If chkMsats.checked then begin
    MicrosatModel(k,N,GrowthStart,GrowthEnd,NumberSimulations);
  end
  else begin

    Val(edtMutationRate.Text, MuRate, code);
    Val(edtSeqLength.Text, SeqLength, code);

    If chkInvarSites.checked then begin
      Val(edtI.Text, PropInVar, code);
      SeqLengthReal := SeqLength-(Seqlength*(PropInVar));
      SeqLength := Trunc(SeqLengthReal);
    end;

    If chkShapeParameter.checked then begin
      Val(edtAlpha.Text, alpha, code);
    end;

    If chkDiploid.checked then begin
      N := 2*N;
    end;

    GeneMuRate(MuRate, SeqLength);
    kSamples := k;

    If chkSeqInput.Checked then begin
      AssignFile(infile, DirOut + '\' + edtSeqFile.Text);
      Reset(infile);
      m := 0;
      while not(Eof(infile)) do begin
        Read(infile, base);
        m := m+1;
      end;
      Sequence := nil;
      SeqLength := m;
      SetLength(Sequence, SeqLength);
      Reset(infile);
      for p := 0 to m-1 do begin
        Read(infile, base);
        Sequence[p] := base;
```

```
      end;
      CloseFile(infile);
      SeqLength := m-1;
    end;

  //currently if we run multiple sims, each sim starts with the same randomly
generated sequence or input sequence provided.
  //we may want to change the random starting sequence each sim, but i'm not sure
why this would be more appropriate than
  //the current model?

    If (chkSeqInput.Checked = False) then begin
      Sequence := nil;
      SetLength(Sequence, SeqLength);

      If (radF81.Checked) or (radHKY85.Checked) then begin
        Val(edtFreqA.Text, FreqA, code);
        Val(edtFreqC.Text, FreqC, code);
        Val(edtFreqG.Text, FreqG, code);
        FreqT := 1-(FreqA+FreqC+FreqG);
        for h := 0 to SeqLength-1 do begin
          RandomChar := Random(100);
          if (RandomChar >= 0) and (RandomChar < FreqA*100) then
            Sequence[h] := 'a';
          if (RandomChar >= FreqA*100) and (RandomChar < (FreqA+FreqC)*100)
then
            Sequence[h] := 'c';
          if (RandomChar >= (FreqA+FreqC)*100) and (RandomChar <
(FreqA+FreqC+FreqG)*100) then
            Sequence[h] := 'g';
          if (RandomChar >= (FreqA+FreqC+FreqG)*100) and (RandomChar <= 100)
then
            Sequence[h] := 't';
        end;
      end
      else begin
      for h := 0 to SeqLength-1 do begin
        RandomChar := Random(4);
        if (RandomChar >= 0) and (RandomChar < 1) then
          Sequence[h] := 'a';
        if (RandomChar >= 1) and (RandomChar < 2) then
          Sequence[h] := 'c';
        if (RandomChar >= 2) and (RandomChar < 3) then
          Sequence[h] := 'g';
        if (RandomChar >= 3) and (RandomChar <= 4) then
          Sequence[h] := 't';
        end;
      end;
    end;
```

```
AssignFile(NewickTreeFile, DirOut + '\' + edtNewickTrees.Text);
Rewrite(NewickTreeFile);

AssignFile(TreeMatrixFile, DirOut + '\' + 'TreeMatFile.txt');
Rewrite(TreeMatrixFile);
AssignFile(MutationMatrixFile, DirOut + '\' + 'MutMatFile.txt');
Rewrite(MutationMatrixFile);

Tmrca := nil;
SetLength(Tmrca, NumberSimulations);
InfAlleles := nil;
SetLength(InfAlleles, NumberSimulations);
FinAlleles := nil;
SetLength(FinAlleles, NumberSimulations);
GammaPerSiteAr := nil;
SetLength(GammaPerSiteAr, NumberSimulations);
GammaGeneMuAr := nil;
SetLength(GammaGeneMuAr, NumberSimulations);
AllelicDiversity := nil;
SetLength(AllelicDiversity, NumberSimulations);
NucleotideDiversity := nil;
SetLength(NucleotideDiversity, NumberSimulations);
NumberMutationArray := nil;
SetLength(NumberMutationArray, NumberSimulations);

for i :=1 to NumberSimulations do begin
  frmRandCoal.Repaint;
  memSimNum.Clear;
  memSimNum.Lines.Add(IntToStr(i));
  CountCoalesce := 0;
  CountCoalesce2 := 0;
  CountMutation := 0;
  CoalesceTime := 0;
  CoalArrayLength := k-1;
  SampleSize := k;
  TreeMatrix := nil;
  MutationMatrix := nil;
  MutationScatter := nil;
  MutLineages := nil;
  MutLineages2 := nil;
  NewickArray := nil;
  NewickArray2 := nil;
  SetLength(TreeMatrix, SampleSize-1, 3);
  SetLength(MutationMatrix, 1000000, 2);
  SetLength(MutationScatter, CoalArrayLength, CoalArrayLength);
  SetLength(MutLineages, CoalArrayLength, k+1);
  SetLength(MutLineages2, CoalArrayLength, k+1);
  SetLength(NewickArray, k+1, k+1);
  SetLength(NewickArray2, k+1, k+1);
  ProgressBar1.FillColor := clblue;
```

```
      ProgressBar1.Max := (k-1)*NumberSimulations;


    //Create identical starting sequences (those before mutation) in an array, with
    //length/rows = # characters and width/columns = # samples. Therefore, character
5
    //in sample 3 will be mutated as Data [4, 2] := ...

      Data := nil;
      SetLength(Data, SeqLength, k);
      for q := 0 to k-1 do begin
        for r := 0 to SeqLength-1 do begin
          Data[r, q] := Sequence[r];
        end;
      end;



      repeat //This repeat loop moves through one generation at a time, asking whether
there is a coalescent or mutation event.
        If (chkFluctuatingN.checked) and (CoalesceTime > GrowthStart) and
(CoalesceTime < GrowthEnd) then begin
          StartN := N;
          Val(edtChangeN.Text, NFactor, code);
          AddInd := ((NFactor/100)*StartN);
          NextNReal := StartN+AddInd;
          NextN := Trunc(NextNReal);
          N := NextN;
        end;

        CoalesceTime := CoalesceTime+1;
        EndofRun := True;
        ChooseRandom(SampleSize,N);
        SortRandom(RandomkArray, SampleSize);
        MakeTree(SortedRandArray, SampleSize);  //CountCoalesce is incremented if
there is a coalescent event
        If chkShapeParameter.checked then
          GammaMutations(SampleSize,SeqLength,i,alpha,MuRate)
        else
        Mutations(SampleSize);
        If CountCoalesce > CountCoalesce2 then begin
          SampleSize := k-CountCoalesce;
          NewickTree(SampleSize, k);
          ProgressBar1.StepBy(1);
        end;
        CountCoalesce2 := CountCoalesce;
        if CountCoalesce < k-1 then
          EndofRun:=False;
      until EndofRun;
```

```
pseudoNewick := NewickArray[k-1, 0];
pseudoNewickArray := nil;
SetLength(pseudoNewickArray, Length(pseudoNewick));


If chkScatterMut.checked then begin
  IdentifyLineageClusters(k);
  MutateLineages(SeqLength);
end;

MutationDistribution(k);
TotalSegSitesFin := 0;
DiversityIndices(SeqLength, k, i);

Append(NewickTreeFile);
TrueNewickTree(k);
StrValue := NewickArray2[k-1,0];
Writeln(NewickTreeFile, StrValue + ';');
CloseFile(NewickTreeFile);

If chkScatterMut.checked and chkSeqOut.Checked then begin


  //If the gene is a diploid gene we must randomly scramble the sequences
  //to assemble genotypes. Else if we leave it ordered genotypes will
  //be arranged as in the genealogy, and alleles within a genotype will
  //more closely related than alleles between genotypes. The same applies
  //to microsat data


  If chkDiploid.Checked then begin
    RandomChosen := nil;
    SetLength(RandomChosen, k);
    rs := 0;
    repeat
      RandomSequence := Random(k+1);
      RandomSequenceInt := Trunc(RandomSequence);
      AlreadyUsed := False;
      rt := 0;
      repeat
        tempRandom := RandomChosen[rt];
        if (tempRandom = RandomSequenceInt) then
          AlreadyUsed := True;
        rt := rt+1;
      until (AlreadyUsed = True) or (rt = rs+1);
      if (AlreadyUsed = False) then begin
        RandomChosen[rs] := RandomSequenceInt;
        rs := rs+1;
      end;
    until (rs = k);
```

```
        If rdgOutput.Items[rdgOutput.ItemIndex] = 'Fasta' then begin
          Append(outfileSeq);
          Writeln(outfileSeq, IntToStr(k) + ' ' + IntToStr(SeqLength));
          Individ := 1;
          for v := 0 to k-1 do begin
            RandomInd := RandomChosen[v];
            if (Odd(v) = False) and (v <> 0) then
              Individ := Individ+1;
            StringName := IntToStr(Individ) + '_' + IntToStr(i) + ' ';
            Write(outfileSeq, StringName:10);
            for w := 0 to SeqLength-1 do begin
              base := Data[w,RandomInd-1];
              Write(outfileSeq, UpperCase(base));
            end;
            Writeln(outfileSeq);
          end;
          WriteLn(outfileSeq);
          CloseFile(outfileSeq);
        end;

        If rdgOutput.Items[rdgOutput.ItemIndex] = 'Clustal' then begin
          Append(outfileSeq);
          Individ := 1;
          for v := 0 to k-1 do begin
            RandomInd := RandomChosen[v];
            if (Odd(v) = False) and (v <> 0) then
              Individ := Individ+1;
            Writeln(outfileSeq, '>' + 'Ind_' + IntToStr(Individ)+ '_' +
IntToStr(RandomInd)+'_'+IntToStr(i));
            for w := 0 to SeqLength-1 do begin
              base := Data[w,RandomInd-1];
              Write(outfileSeq, UpperCase(base));
            end;
            Writeln(outfileSeq);
          end;
          WriteLn(outfileSeq);
          CloseFile(outfileSeq);
        end;

        If rdgOutput.Items[rdgOutput.ItemIndex] = 'MrBayes' then begin
          Append(outfileSeq);
          Individ := 1;
          Writeln(outfileSeq,'#NEXUS');
          Writeln(outfileSeq);
          Writeln(outfileSeq,'BEGIN DATA;');
          Writeln(outfileSeq, 'DIMENSIONS ' + 'NTAX=' + IntToStr(k)+ ' NCHAR='
+ IntToStr(SeqLength) + ';');
          Writeln(outfileSeq, 'FORMAT DATATYPE=DNA;');
          Writeln(outfileSeq, 'MATRIX');
```

```
            for v := 0 to k-1 do begin
              RandomInd := RandomChosen[v];
              if (Odd(v) = False) and (v <> 0) then
                Individ := Individ+1;
              Writeln(outfileSeq, 'Ind' + IntToStr(Individ) + '_' +
IntToStr(RandomInd)+'_'+IntToStr(i));
              for w := 0 to SeqLength-1 do begin
                base := Data[w,RandomInd-1];
                Write(outfileSeq, UpperCase(base));
              end;
              Writeln(outfileSeq);
            end;
            Writeln(outfileSeq, ';');
            Writeln(outfileSeq, 'end;');
            WriteLn(outfileSeq);
            Writeln(outfileSeq, 'begin mrbayes;');
            Writeln(outfileSeq, 'set autoclose=yes;');
            if radJC69.Checked then
              Nst := 1
            else Nst := 2;
            if chkShapeParameter.Checked then
              StringShape := 'rates=gamma'
            else StringShape := '';
            Writeln(outfileSeq, 'lset nst=' + IntToStr(Nst) + ' ' + StringShape + ';');
            Writeln(outfileSeq, 'mcmc ngen=1000 printfreq=100 samplefreq=10
nchains=4 savebrlens=yes;');
            Writeln(outfileSeq, 'end;');
            CloseFile(outfileSeq);
          end;

          If rdgOutput.Items[rdgOutput.ItemIndex] = 'Nexus' then begin
            Append(outfileSeq);
            Individ := 1;
            Writeln(outfileSeq,'#NEXUS');
            Writeln(outfileSeq);
            Writeln(outfileSeq,'BEGIN DATA;');
            Writeln(outfileSeq, 'DIMENSIONS ' + 'NTAX=' + IntToStr(k)+ ' NCHAR='
+ IntToStr(SeqLength) + ';');
            Writeln(outfileSeq, 'FORMAT DATATYPE=DNA;');
            Writeln(outfileSeq, 'MATRIX');
            for v := 0 to k-1 do begin
              RandomInd := RandomChosen[v];
              if (Odd(v) = False) and (v <> 0) then
                Individ := Individ+1;
              Writeln(outfileSeq, 'Ind_' + IntToStr(Individ) + '_' +
IntToStr(RandomInd)+'_'+IntToStr(i));
              for w := 0 to SeqLength-1 do begin
                base := Data[w,RandomInd-1];
                Write(outfileSeq, UpperCase(base));
              end;
```

```
        Writeln(outfileSeq);
      end;
      Writeln(outfileSeq, ';');
      Writeln(outfileSeq, 'end;');
      WriteLn(outfileSeq);
      CloseFile(outfileSeq);
    end;

    If chkArlequin.Checked then begin
      Append(ArlequinBatch);
      Individ := 1;
      FileCounter := DirOut + '/Arlequin/sim_' + IntToStr(i) + '.arp';
      Writeln(ArlequinBatch, FileCounter);
      CloseFile(ArlequinBatch);
      AssignFile(ArlequinProject, FileCounter);
      Rewrite(ArlequinProject);
      Writeln(ArlequinProject, '[Profile]');
      Writeln(ArlequinProject, '   Title="DNA sequence data generated using
coalescent simulations"');
      Writeln(ArlequinProject, '   NbSamples=1');
      Writeln(ArlequinProject, '   GenotypicData=1');
      Writeln(ArlequinProject, '   DataType=DNA');
      Writeln(ArlequinProject, '   LocusSeparator=NONE');
      Writeln(ArlequinProject, '[Data]');
      Writeln(ArlequinProject, ' [[Samples]]');
      Writeln(ArlequinProject, '   SampleName="Homogenous population"');
      ArlequinSampleSize := IntToStr(k div 2);
      Writeln(ArlequinProject, '   SampleSize=' + ArlequinSampleSize);
      Writeln(ArlequinProject, '   SampleData= {');

      for ai := 0 to k-1 do begin
        RandomInd := RandomChosen[ai];
        if (Odd(ai) = False) and (ai <> 0) then
          Individ := Individ+1;
        if (odd(ai) = False) or (ai = 0) then begin
          StringName := IntToStr(Individ) + ' ';
          Write(ArlequinProject, StringName: 10);
          Write(ArlequinProject, ' 1 ');
        end
        else begin
          Write(ArlequinProject, '            ');
        end;
        for aj := 1 to SeqLength do begin
          valueChar := Data[aj-1, RandomInd-1];
          Write(ArlequinProject, UpperCase(valueChar));
        end;
        Writeln(ArlequinProject);
      end;
      Writeln(ArlequinProject, '}');
      CloseFile(ArlequinProject);
```

```
  end;
 end

 else begin
  If rdgOutput.Items[rdgOutput.ItemIndex] = 'Fasta' then begin
   Append(outfileSeq);
   Writeln(outfileSeq, IntToStr(k) + ' ' + IntToStr(SeqLength));
   for v := 0 to k-1 do begin
    StringName := IntToStr(v+1) + '_' + IntToStr(i) + ' ';
    Write(outfileSeq, StringName:10);
    for w := 0 to SeqLength-1 do begin
     base := Data[w,v];
     Write(outfileSeq, UpperCase(base));
    end;
    Writeln(outfileSeq);
   end;
   WriteLn(outfileSeq);
   CloseFile(outfileSeq);
  end;

  If rdgOutput.Items[rdgOutput.ItemIndex] = 'Clustal' then begin
   Append(outfileSeq);
   for v := 0 to k-1 do begin
    Writeln(outfileSeq, '>' + IntToStr(v+1)+'_'+IntToStr(i));
    for w := 0 to SeqLength-1 do begin
     base := Data[w,v];
     Write(outfileSeq, UpperCase(base));
    end;
    Writeln(outfileSeq);
   end;
   WriteLn(outfileSeq);
   CloseFile(outfileSeq);
  end;

  If rdgOutput.Items[rdgOutput.ItemIndex] = 'MrBayes' then begin
   Append(outfileSeq);
   Writeln(outfileSeq,'#NEXUS');
   Writeln(outfileSeq);
   Writeln(outfileSeq,'BEGIN DATA;');
   Writeln(outfileSeq, 'DIMENSIONS ' + 'NTAX=' + IntToStr(k)+ ' NCHAR='
+ IntToStr(SeqLength) + ';');
   Writeln(outfileSeq, 'FORMAT DATATYPE=DNA;');
   Writeln(outfileSeq, 'MATRIX');
   for v := 0 to k-1 do begin
    Writeln(outfileSeq, IntToStr(v+1)+'_'+IntToStr(i));
    for w := 0 to SeqLength-1 do begin
     base := Data[w,v];
     Write(outfileSeq, UpperCase(base));
    end;
    Writeln(outfileSeq);
```

```
            end;
            Writeln(outfileSeq, ';');
            Writeln(outfileSeq, 'end;');
            WriteLn(outfileSeq);
            Writeln(outfileSeq, 'begin mrbayes;');
            Writeln(outfileSeq, 'set autoclose=yes;');
            if radJC69.Checked then
              Nst := 1
            else Nst := 2;
            if chkShapeParameter.Checked then
              StringShape := 'rates=gamma'
            else StringShape := '';
            Writeln(outfileSeq, 'lset nst=' + IntToStr(Nst) + ' ' + StringShape + ';');
            Writeln(outfileSeq, 'mcmc ngen=1000 printfreq=100 samplefreq=10
nchains=4 savebrlens=yes;');
            Writeln(outfileSeq, 'end;');
            CloseFile(outfileSeq);
          end;

          If rdgOutput.Items[rdgOutput.ItemIndex] = 'Nexus' then begin
            Append(outfileSeq);
            Writeln(outfileSeq,'#NEXUS');
            Writeln(outfileSeq);
            Writeln(outfileSeq,'BEGIN DATA;');
            Writeln(outfileSeq, 'DIMENSIONS ' + 'NTAX=' + IntToStr(k)+ ' NCHAR='
+ IntToStr(SeqLength) + ';');
            Writeln(outfileSeq, 'FORMAT DATATYPE=DNA;');
            Writeln(outfileSeq, 'MATRIX');
            for v := 0 to k-1 do begin
              Writeln(outfileSeq, IntToStr(v+1)+'_'+IntToStr(i));
              for w := 0 to SeqLength-1 do begin
                base := Data[w,v];
                Write(outfileSeq, UpperCase(base));
              end;
              Writeln(outfileSeq);
            end;
            Writeln(outfileSeq, ';');
            Writeln(outfileSeq, 'end;');
            WriteLn(outfileSeq);
            CloseFile(outfileSeq);
          end;

          If chkArlequin.Checked then begin
            Append(ArlequinBatch);
            FileCounter := DirOut + '/Arlequin/sim_' + IntToStr(i) + '.arp';
            Writeln(ArlequinBatch, FileCounter);
            CloseFile(ArlequinBatch);
            AssignFile(ArlequinProject, FileCounter);
            Rewrite(ArlequinProject);
            Writeln(ArlequinProject, '[Profile]');
```

```
          Writeln(ArlequinProject, '   Title="DNA sequence data generated using
coalescent simulations"');
          Writeln(ArlequinProject, '   NbSamples=1');
          Writeln(ArlequinProject, '   GenotypicData=0');
          Writeln(ArlequinProject, '   DataType=DNA');
          Writeln(ArlequinProject, '   LocusSeparator=NONE');
          Writeln(ArlequinProject, '[Data]');
          Writeln(ArlequinProject, ' [[Samples]]');
          Writeln(ArlequinProject, '   SampleName="Panmictic population"');
          ArlequinSampleSize := IntToStr(k);
          Writeln(ArlequinProject, '   SampleSize=' + ArlequinSampleSize);
          Writeln(ArlequinProject, '   SampleData= {');

          for ai := 0 to k-1 do begin
            StringName := IntToStr(ai+1) + ' ';
            Write(ArlequinProject, StringName:10);
            Write(ArlequinProject, ' 1 ');
            for aj := 1 to SeqLength do begin
              valueChar := Data[aj-1, ai];
              Write(ArlequinProject, UpperCase(valueChar));
            end;
            Writeln(ArlequinProject);
          end;
          Writeln(ArlequinProject, '}');
          CloseFile(ArlequinProject);
        end;

      end;


    end;

    value := TreeMatrix[k-2, 0];
    Tmrca[i-1] := value;
    InfAlleles[i-1] := SegSites;
    FinAlleles[i-1] := TotalSegSitesFin;

    Append(TreeMatrixFile);
    Writeln(TreeMatrixFile, '#' + IntToStr(i));
    for tmf2 := 0 to k-2 do begin
      for tmf := 0 to 2 do begin
        value := TreeMatrix[tmf2, tmf];
        Write(TreeMatrixFile, IntToStr(value) + ' ');
      end;
      Writeln(TreeMatrixFile);
    end;
    CloseFile(TreeMatrixFile);

    Append(MutationMatrixFile);
    Writeln(MutationMatrixFile, '#' + IntToStr(i));
```

```
        for mmf := 0 to CountMutation-1 do begin
          for mmf2 := 0 to 1 do begin
            value := MutationMatrix[mmf,mmf2];
            Write(MutationMatrixFile, IntToStr(value) + ' ');
          end;
          Writeln(MutationMatrixFile);
        end;
        CloseFile(MutationMatrixFile);
        NumberMutationArray[i-1] := CountMutation;


      end; // end of for i:=1 to NumberSimulations loop
      Outputs(NumberSimulations,0,k);
      TreeCounter := NumberSimulations;
      DrawTree;
    end;  // end of sequence model loop
    EndTime := Time;
    LogFile(k,N,NBeforeGrowth,SeqLength,GrowthStart,GrowthEnd,
NumberSimulations, StartTime, EndTime, MuRate,FreqA,FreqC,FreqG,FreqT);

    SimDone := True;
  end  //end of if (k>0) and (N>0) loop
  else begin
    resultString := 'Please insert Sample and Population Sizes';
    Application.MessageBox(PChar(resultString), 'WARNING', [smbok]);
  end;

end;

procedure TfrmRandCoal.SaveGenealogy(Sender: TObject);

var
 resultString : string;

begin
  if (SimDone = False) then begin
    resultString := 'Please run simulations first';
    Application.MessageBox(PChar(resultString), 'WARNING', [smbok]);
  end
  else begin
    dlgSaveFile.Execute;
    Image1.Picture.Bitmap.SaveToFile(dlgSaveFile.FileName);
  end;
end;

procedure TfrmRandCoal.CreateForm(Sender: TObject);

begin
  Image3.Picture.LoadFromFile('./img/meep_logo.png');
  SimDone := False;
```

```
  DirOut := '.';
end;

procedure TfrmRandCoal.AssignDirectory(Sender: TObject);

begin
  MemDir.Clear;
  DirOut := DirectoryTreeView1.Directory;
  MemDir.Lines.Add(DirOut);
end;

procedure TfrmRandCoal.TreeFwd(Sender: TObject);

var
  TestString : string;
  ReadString : string;
  ReadChar : string;
  ReadInt : integer;
  i,j : integer;
  k,code : integer;
  resultString : string;
  NLoci : integer;
  SimNumber, LocusNumber : integer;

begin
  if (SimDone = False) then begin
    resultString := 'Please run simulations first';
    Application.MessageBox(PChar(resultString), 'WARNING', [smbok]);
  end
  else begin
    Val(edtSampleSize.Text, k, code);
    Val(edtNumLoci.Text, NLoci, code);
    if chkMSats.Checked then begin
      if TreeCounter = NumberSimulations*NLoci then
        TreeCounter := 1
      else
        TreeCounter := TreeCounter + 1;
      AssignFile(TreeMatrixFile, DirOut + '\' + 'TreeMatFile.txt');
      Reset(TreeMatrixFile);
      AssignFile(MutationMatrixFile, DirOut + '\' + 'MutMatFile.txt');
      Reset(MutationMatrixFile);
      TestString := '#' + IntToStr(TreeCounter);

      repeat
        Read(TreeMatrixFile, ReadString);
        if (ReadString = TestString) then begin

          for i := 0 to k-2 do begin
            for j := 0 to 2 do begin
              Read(TreeMatrixFile, ReadInt);
```

```
            TreeMatrix[i,j] := ReadInt;
          end;
          Readln(TreeMatrixFile);
        end;
      end
      else
        Readln(TreeMatrixFile);
    until (ReadString = TestString);

    MutationMatrix := nil;
    CountMutation := NumberMutationArray[TreeCounter-1];
    SetLength(MutationMatrix, CountMutation, 2);
    repeat
      Read(MutationMatrixFile, ReadString);
      if (ReadString = TestString) then begin

        for i := 0 to CountMutation-1 do begin
          for j := 0 to 1 do begin
            Read(MutationMatrixFile, ReadInt);
            MutationMatrix[i,j] := ReadInt;
          end;
          Readln(MutationMatrixFile);
        end;
      end
      else
        Readln(MutationMatrixFile);
    until (ReadString = TestString);

    DrawTree;

    LocusNumber := TreeCounter mod NLoci;
    SimNumber := Ceil(TreeCounter/NLoci);
    if LocusNumber = 0 then
      LocusNumber := TreeCounter div SimNumber;

    with Image1,Canvas do begin
      Font.Color := clBlack;
      Font.Size := 10;
      Textout(10, 510, 'Microsatellite Locus ' + IntToStr(LocusNumber) + ' Simulation
' + IntToStr(SimNumber));
    end;
  end

  else begin
    if TreeCounter = NumberSimulations then
      TreeCounter := 1
    else
    TreeCounter := TreeCounter + 1;
    AssignFile(TreeMatrixFile, DirOut + '\' + 'TreeMatFile.txt');
    Reset(TreeMatrixFile);
```

```
    AssignFile(MutationMatrixFile, DirOut + '\' + 'MutMatFile.txt');
    Reset(MutationMatrixFile);
    TestString := '#' + IntToStr(TreeCounter);

    repeat
      Read(TreeMatrixFile, ReadString);
      if (ReadString = TestString) then begin

        for i := 0 to k-2 do begin
          for j := 0 to 2 do begin
            Read(TreeMatrixFile, ReadInt);
            TreeMatrix[i,j] := ReadInt;
          end;
          Readln(TreeMatrixFile);
        end;
      end
      else
        Readln(TreeMatrixFile);
    until (ReadString = TestString);

    MutationMatrix := nil;
    CountMutation := NumberMutationArray[TreeCounter-1];
    SetLength(MutationMatrix, CountMutation, 2);
    repeat
      Read(MutationMatrixFile, ReadString);
      if (ReadString = TestString) then begin

        for i := 0 to CountMutation-1 do begin
          for j := 0 to 1 do begin
            Read(MutationMatrixFile, ReadInt);
            MutationMatrix[i,j] := ReadInt;
          end;
          Readln(MutationMatrixFile);
        end;
      end
      else
        Readln(MutationMatrixFile);
    until (ReadString = TestString);

    DrawTree;
    with Image1,Canvas do begin
      Font.Color := clBlack;
      Font.Size := 10;
      Textout(10, 510, IntToStr(TreeCounter) + '\' + IntToStr(NumberSimulations));
    end;
  end;
 end;
 CloseFile(TreeMatrixFile);
 CloseFile(MutationMatrixFile);
end;
```

```
procedure TfrmRandCoal.TreeBack(Sender: TObject);

var
  TestString : string;
  ReadString : string;
  ReadInt : integer;
  i,j,k,code : integer;
  resultString : string;
  NLoci : integer;
  LocusNumber, SimNumber : integer;


begin
  if (SimDone = False) then begin
    resultString := 'Please run simulations first';
    Application.MessageBox(PChar(resultString), 'WARNING', [smbok]);
  end
  else begin
    Val(edtSampleSize.Text, k, code);
    Val(edtNumLoci.Text, NLoci, code);
    if chkMSats.Checked then begin
      if TreeCounter = 1 then
        TreeCounter := NumberSimulations*NLoci
      else
        TreeCounter := TreeCounter - 1;
      AssignFile(TreeMatrixFile, DirOut + '\' + 'TreeMatFile.txt');
      Reset(TreeMatrixFile);
      AssignFile(MutationMatrixFile, DirOut + '\' + 'MutMatFile.txt');
      Reset(MutationMatrixFile);
      TestString := '#' + IntToStr(TreeCounter);

      repeat
        Read(TreeMatrixFile, ReadString);
        if (ReadString = TestString) then begin

          for i := 0 to k-2 do begin
            for j := 0 to 2 do begin
              Read(TreeMatrixFile, ReadInt);
              TreeMatrix[i,j] := ReadInt;
            end;
            Readln(TreeMatrixFile);
          end;
        end
        else
          Readln(TreeMatrixFile);
      until (ReadString = TestString);

      MutationMatrix := nil;
      CountMutation := NumberMutationArray[TreeCounter-1];
```

```
    SetLength(MutationMatrix, CountMutation, 2);
    repeat
      Read(MutationMatrixFile, ReadString);
      if (ReadString = TestString) then begin

        for i := 0 to CountMutation-1 do begin
          for j := 0 to 1 do begin
            Read(MutationMatrixFile, ReadInt);
            MutationMatrix[i,j] := ReadInt;
          end;
          Readln(MutationMatrixFile);
        end;
      end
      else
        Readln(MutationMatrixFile);
    until (ReadString = TestString);

    DrawTree;

    LocusNumber := TreeCounter mod NLoci;
    SimNumber := Ceil(TreeCounter/NLoci);
    if LocusNumber = 0 then
      LocusNumber := TreeCounter div SimNumber;


    with Image1,Canvas do begin
      Font.Color := clBlack;
      Font.Size := 10;
      Textout(10, 510, 'Microsatellite Locus ' + IntToStr(LocusNumber) + ' Simulation
' + IntToStr(SimNumber));
    end;
  end

  else begin
    if TreeCounter = 1 then
      TreeCounter := NumberSimulations
    else
    TreeCounter := TreeCounter - 1;
    AssignFile(TreeMatrixFile, DirOut + '\' + 'TreeMatFile.txt');
    Reset(TreeMatrixFile);
    AssignFile(MutationMatrixFile, DirOut + '\' + 'MutMatFile.txt');
    Reset(MutationMatrixFile);
    TestString := '#' + IntToStr(TreeCounter);

    repeat
      Read(TreeMatrixFile, ReadString);
      if (ReadString = TestString) then begin

        for i := 0 to k-2 do begin
          for j := 0 to 2 do begin
```

```
            Read(TreeMatrixFile, ReadInt);
            TreeMatrix[i,j] := ReadInt;
          end;
          Readln(TreeMatrixFile);
        end;
      end
      else
        Readln(TreeMatrixFile);
    until (ReadString = TestString);

    MutationMatrix := nil;
    CountMutation := NumberMutationArray[TreeCounter-1];
    SetLength(MutationMatrix, CountMutation, 2);
    repeat
      Read(MutationMatrixFile, ReadString);
      if (ReadString = TestString) then begin

        for i := 0 to CountMutation-1 do begin
          for j := 0 to 1 do begin
            Read(MutationMatrixFile, ReadInt);
            MutationMatrix[i,j] := ReadInt;
          end;
          Readln(MutationMatrixFile);
        end;
      end
      else
        Readln(MutationMatrixFile);
    until (ReadString = TestString);

    DrawTree;
    with Image1,Canvas do begin
      Font.Color := clBlack;
      Font.Size := 10;
      Textout(10, 510, IntToStr(TreeCounter) + '\' + IntToStr(NumberSimulations));
    end;
  end;
 end;
 CloseFile(TreeMatrixFile);
 CloseFile(MutationMatrixFile);
end;

end.
```