Mathematical Stochastic Models for DNA

Thesis Submitted in Partial Fulfillment

of the Requirements

of the Jay and Jeanie Schottenstein Honors Program

Yeshiva College

Yeshiva University

May 2023

Yonah Moise

Mentor: Professor Peter Nandori, Mathematics

Mathematical Stochastic Models for DNA

An Honors Thesis in Partial Fulfillment of the Requirements for the Yeshiva University Jay and Judy Schottenstein Honors Program, May 2023, under the supervision of Professor Peter Nandori

Yonah Moise[1]

[1]Yeshiva University, Department of Mathematics

Due May 2023

Abstract

Kimura's neutral theory of molecular evolution gave rise to several models by which to study genetic data. Such models include the infinite alleles model and the infinite sites model. We studied these models, and present them here in an clear and algorithmic style. We coded these models in Python, using Monte Carlo methods to calculate probabilities and perform tests of hypothesis against real-world data.

Contents

1 Background

With the availability of new data, questions regarding expectation, variance, distribution, etc. of protein and genetic polymorphisms began to arise. Back in the 1960s, molecular polymorphism data began to be produced. Researchers ran-out different enzymes on gels and, observing different mobilities, classified them into different types. The observed distributions of these data could not be predicted or explained by classical population genetics. Thus, new models were needed to answer questions such as: what should we expect to see? how many alleles? what level of heterozygosity? Furthermore, in the 1980s, genetic sequence polymorphisms began to be examined. New questions started to arise, such as: what is the distribution along the genome? Looking at pairwise differences between sequences, does it matter if we sample from the same individual or from different individuals? Does it matter if we sample from the same geographic location or from different locations? What about differences between species? None of the dynamic systems models which existed could help with these questions.

One such new model that was created was Kimura's neutral theory of molecular evolution, which posits that most genetic variation is due to mutations and genetic drift. The view of Kimura's theory is basically that genetic polymorphism data represent the outcome of a single, highly complex, non-repeatable evolutionary history. Now, since (according to Kimura) the pattern of genetic variation is stochastic, stochastic processes were needed to analyse this. However, the inherent problem with this approach is in trying to interpret the single outcome of a stochastic process. This is inherently different from experiments where replication is possible; for a stochastic process, the probability of exact replication is very small.

The stochastic process known as 'the coalescent' presents a coherent statistical framework for analyzing genetic polymorphism data. It's a way of taking Kimura's diffusion theory and running it backwards in time instead. Indeed, the coalescent has played a central role in

population genetics for well over 30 years.

Coalescent models follow the genealogy (ancestry) of genes backward in time, starting from the present. This turns out to be a very powerful way of thinking about genetic polymorphism. It leads to elegant mathematics, powerful simulation algorithms (because instead of simulating the whole population forwards in time, you can just simulate the ancestry of the things you actually sampled backward in time).[1]

But to run models algorithmically, and use the data generated by stochastic processes to evaluate real-world genetic data, we have to run models forwards in time. Regarding the work done for this thesis, we have used Monte Carlo simulations (based on simulations of infinite alleles model and infinite sites model, which produce synthetic 'genetic' data for our purposes) to analyze real genetic data.

1.1 Notation to Describe an Allelic Partition

We presently explain the two salient notational terms used to describe an allelic partition. The two elements are j and a_j. The term j describes how many times an allele appears in the sample population. And the term a_j describes how many alleles appear j times in the sample population. Thus, adding up all the a_j terms gives us the total number of alleles found in the sample population. Furthermore, multiplying each a_j term by its corresponding j term, and then summing all these products, gives us the size of the sample population. In mathematical terms: $n = \sum_{j=0}^{n} j \cdot a_j$.

1.1.1 Example

Let us turn to the data found in Coyne's 1976 study to serve as an example through which to get used to this notation. Coyne found "23 alleles in 60 family lines at the xanthine dehydrogenase locus of *Drosophila persimilis*" [D08, page 15] with the following allelic partition:

$$a_1 = 18,\ a_2 = 3,\ a_4 = 1,\ a_{32} = 1.$$

If we sum the a_j terms, we get 23. Let us reproduce this operation explicitly: $18 + 3 + 1 + 1 = 23$. Furthermore, the operation $\sum_{j=0}^{n} j \cdot a_j$ does indeed produce 60. Let us reproduce the salient parts of this operation explicitly: $18 \cdot 1 + 3 \cdot 2 + 1 \cdot 4 + 1 \cdot 32 = 18 + 6 + 4 + 32 = 60$.

2 Infinite Alleles Model

2.1 Infinite Alleles Assumption

We assume that so many possible alleles exist that each new mutation produces a new allele. Kimura explained the rationale for this assumption by arguing that "if a gene consists of 500

[1] The material in this section up to this point is heavily quoting or paraphrasing from a 2018 lecture by Magnus Nordborg [N18]. The quotation marks have been removed in this section for improved readability.

nucleotides, the number of possible DNA sequences is $4^{500} = 10^{\log_{10} 4 \cdot 500} = 10^{500 \cdot \frac{\log 4}{\log 10}} \approx 10^{301}$. Since a single-base mutation has the potential to change the original-sequence to $3 \cdot 500 = 1500$ of the 4^{500} sequences, the probability of reverting to the original-sequence with a second single-base mutation is $\frac{1}{1500}$ – and "thus, the total number of possible alleles is essentially infinite" [D08, page 14].

2.2 Hoppe's Urn Model

We begin with a mutation-opportunity of weight θ. ($\theta = 4N \cdot \mu$ is related to the mutation rate μ [D08, page 10].) Since this is the only option, we select the mutation-opportunity and our first allele (with a weight of 1) is produced. Now we can select either the mutation-opportunity (to produce a new-allele) or a copy of this allele (to produce another offspring, with weight 1, of this allele). As the process progresses, and there is a greater variety of different alleles, we say that either the mutation-opportunity is selected or some copy of some allele is selected.

When a new-allele is created, we represent it and its descendants in a new tree. Selecting a copy of some allele in the Hoppe's Urn (HU) process corresponds to a coalescent event – that is, a branching on a tree.

At each moment in time, some event occurs (either a mutation event or a coalescent event) – which is determined by following the uniform distribution. Starting at time $k = 0$, mutations occurs at rate $\frac{\theta}{\theta+k}$, and coalescent events occur with probability $\frac{k}{\theta+k}$. After the final time (i.e. the final event), there are $n = k + 1$ lineages/descendants [D08, page 16].

Below, in figure 2.1, we present a picture which depicts a forest of trees after a run of the HU-model:

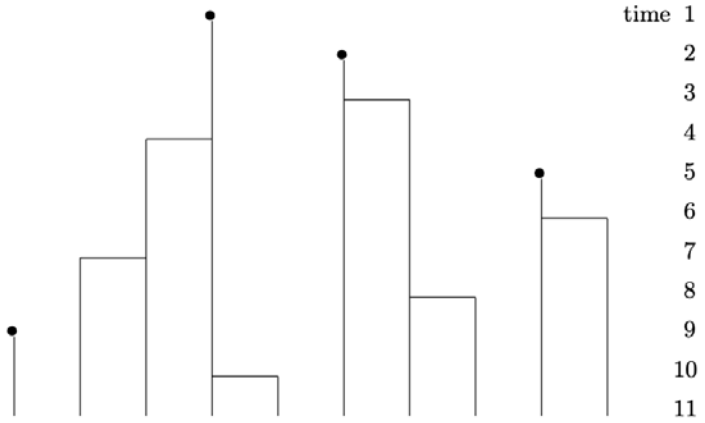

Figure 2.1: Example of a Hoppe's Urn Forest [D08, page 16].

2.2.1 Chinese Restaurant Process

Joyce and Tavaré (1987) "added bookkeeping to keep track of the history" of an HU-process [D08, page 20]. By viewing the historical process as a permutation, we construct a "cycle decomposition" to store the historical data (which may be reconstructed-in-full from the decomposition) [D08, page 21].

We will number the genes by the order that they were produced in the HU-process. A new allele starts a new cycle. A new copy of an allele is inserted into the same cycle, in the left-most position of the cycle.

See figure 2.2 below, where we present part of the Hoppe's Urn process that we depicted above in figure 2.1, as an example of this decomposition:

(1)	1 is always a new color
(1)(2)	2 is a new color
(1)(32)	3 is a child of 2
(41)(32)	4 is a child of 1
(41)(32)(5)	5 is a new color
(41)(32)(65)	6 is a child of 5
(741)(32)(65)	7 is a child of 4
(741)(832)(65)	8 is a child of 3

Figure 2.2: Example of a Chinese Restaurant Process Decomposition [D08, page 21].

2.3 A Sufficient Statistic for θ

A theorem from Watterson (1975) says that $\mathbb{E}(K_n) \sim \theta log(n)$, where K_n is the number of different alleles in the sample population, and n is the size of the sample population [D08, page 17].
Furthermore, we know from Durrett's Theorem 1.13 that K_n is a sufficient statistic for estimating θ [D08, page 22]. Thus, if we know K_n (the number of different alleles), then we have a sufficient statistic by which to estimate θ – that is, we estimate θ with $\frac{K_n}{log(n)}$.

2.4 Sample Homozygosity

Sample homozygosity is "a random variable that gives the probability that in the given sample two randomly chosen members are identical" [D08, page 24]. Watterson (1977) presented a version of the sample homozygosity, which "represents the probability that two individuals chosen with replacement are the same" [D08, page 25-26]. His formula is

$$F_n = \sum_{j=1}^{n} a_j \cdot \left(\frac{j}{n}\right)^2,$$

"where a_j is the number of alleles with j representatives" [D08, page 25]. Thus, this statistic relies on allelic partition data.

2.5 Computer Simulations

In our code, we defined a class of objects called "Study" – where each genetic study and its data is treated and analyzed separately. Input into each Study object is N (the number of simulations), n (sample population size), K_n (number of different alleles or haplotypes), and the F_n value from those genetic data. For this object, we use our data and the infinite alleles model to test whether our null-hypothesis (of neutral theory of molecular evolution) holds.

We wrote a function called "infiniteAlleles()" – which took inputs of N, n, K_n, and the F_n value. The function first calls another function to calculate θ using inputs of n and K_n (using the sufficient statistic described above, in section 3.3). Then Hoppe's Urn simulations are run until N number of simulations are "successful" – that is, for our purposes, that there are N simulations in which K_n number of different alleles are generated.

For all successful simulations, we list the sizes of each tree (i.e. how many representatives each allele has), and make a dictionary which counts how many alleles (trees) have the same number of representatives (descendants). In other words, we calculate the a_j values; and specifically, we create a dictionary in which the keys are j (the number of representatives) and the value-pairs are a_j (the number of trees with with the same number of descendants).

This dictionary (with j and a_j data) is then input into another function, with our sample size number n, that calculates the sample homozygosity (as described above, in section 3.4). After calculating this value (F_n), we add it do a vector which contains the F_n values of all successful simulations – which the "infiniteAlleles()" function returns, and which is stored as a value of an attribute of this Study object.

We also compare this F_n value (from our successful Hoppe's Urn simulation) with the F_n value of the study data (the F_n statistic which is calculated with the j's and a_j's from the study's real-world data). We increment a counter ("larger_counter") each time the simulation-partition's F_n is larger than the study-partition's F_n – thus keeping track of how many times this occurs. We now use this ("larger_counter") to calculate the p-value.

A p-value is "the probability of obtaining results at least as extreme as the observed results of a statistical hypothesis test, assuming that the null hypothesis is correct." It "serves... to provide the smallest level of significance at which the null hypothesis would be rejected. A smaller p-value means that there is stronger evidence in favor of the alternative hypothesis" [B23]. For us, the p-value will give us a level of the F_n values for rejecting the null-hypothesis. Thus, if the F_n from the study's real-world data is larger than the p-value, then we reject the null-hypothesis – and say that fitness of the alleles is a factor in the evolution of the gene pool.

We calculate the p-value with the following operation: $\frac{larger\_counter}{N}$. This is the number of times the successful-simulation-partition's F_n is larger than the study-partition's F_n, over

the total number of "successful" simulations.

2.5.1 Simulation with *Drosophila persimilis* Genetic Data from Coyne (1976)

Coyne's study found "23 alleles in 60 family lines at the xanthine dehydrogenase locus of *Drosophila persimilis*" [D08, page 15] with the following allelic partition:

$$a_1 = 18,\ a_2 = 3,\ a_4 = 1,\ a_{32} = 1.$$

We created a instance of the Study class for data from Coyne's study of the *Drosophila persimilis*. Watterson (1977) ran $N = 1000$ successful infinite-allele simulations for his computation experiment, while we ran our code for $N = 10,000$ and $N = 100,000$ successful infinite-allele simulations. And Durrett presents 0.2972 as the F_n value of the Coyne data [D08, page 26]. We used this F_n value, and confirmed by our own calculation (based on the allelic partition of Coyne's data which Durrett himself presents) that the F_n value is indeed $107/360 = 0.2972222...$

Below are two histograms, which depict the F_n values produced from the infinite-alleles simulations (on the x-axis), against the number of simulations which produced that same F_n value (on the y-axis).

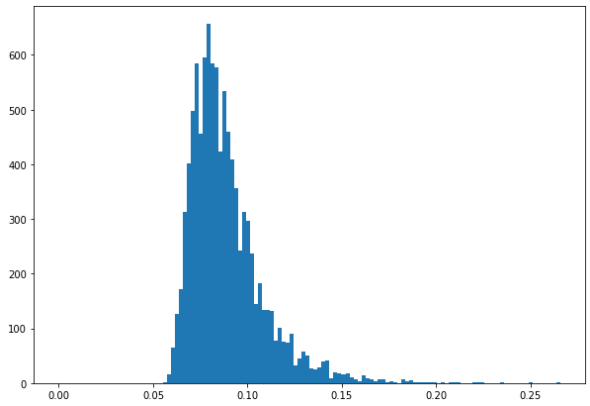

Figure 2.3: Coyne, 10,000 simulations.

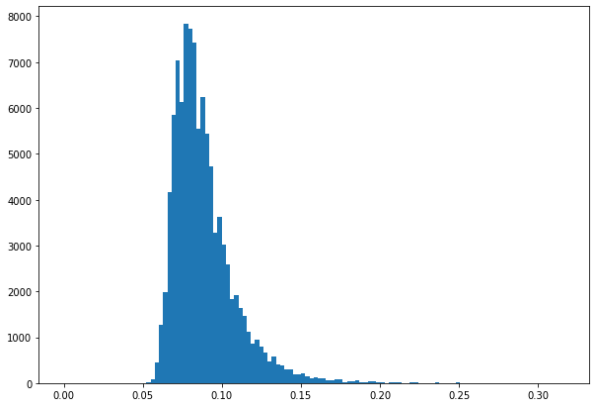

Figure 2.4: Coyne, 100,000 simulations.

Figure 2.3 depicts F_n data from $10,000$ simulations. Zero simulations produced allelic partitions with an F_n value as extreme/high as that of the Coyne data. Thus, by way of a Monte Carlo simulation, we have produced an estimated p-value (the probability of obtaining an F_n value as high as that of the Coyne data) of 0. Figure 2.4 depicts F_n data from $100,000$ simulations. Here, a Monte Carlo simulation produces an estimated p-value of 0.00002 – that is, $\frac{2}{100,000}$ F_n values produced from the allelic partitions of these Hoppe's Urn simulations are as extreme/large as the F_n value produced from the Coyne data. These small p-values are indicative of the Coyne data not fitting the null-hypothesis of neutral evolution.

2.5.2 Simulation with Drosophila pseudo-obscura Genetic Data from Singh, Lewontin, and Felton (1976).

Singh, Lewontin, and Felton's study "found 27 alleles in 146 genes from the xanthine dehydrogenase locus of" *Drosophila pseudo-obscura* [D08, page 15] with the following allelic partition:

$$a_1 = 20,\ a_2 = 3,\ a_3 = 7,\ a_5 = 2,\ a_6 = 2,\ a_8 = 1,\ a_{11} = 1,\ a_{68} = 1.$$

We created a instance of the Study class for data from Singh, Lewontin, and Felton's (SLF) study of the Drosophila pseudo-obscura. Durrett cites the results of a similar process in his book. Durrett (p. 26; seeming to cite Watterson (1977)) ran $N = 2000$ successful infinite-allele simulations for his computation experiment, while we ran our code for $N = 10,000$ and $N = 100,000$ successful infinite-allele simulations. Furthermore, Durrett presents 0.2353 as the F_n value of the SLF data [D08, page 26], and we used this F_n value for consistency with Durrett's simulations. However, our own calculation of the F_n statistic (based on the allelic partition of SLF's data which Durrett himself presents) – which was intended to merely confirm Durrett's value (again, 0.2353) – found an F_n value of $2513/10658 = 0.235785325577031337...$

As before, below are two histograms, which depict the F_n values produced from the infinite-alleles simulations (on the x-axis), against the number of simulations which produced that same F_n value (on the y-axis).

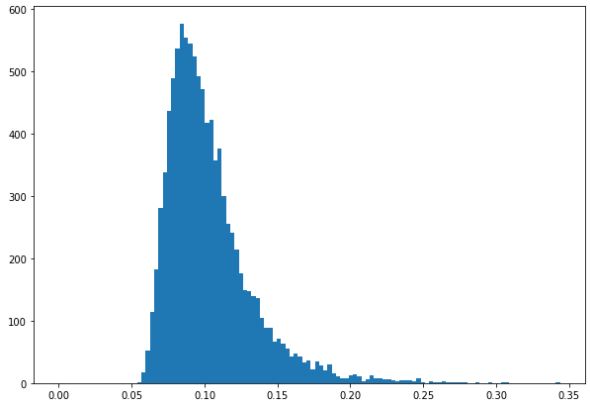

Figure 2.5: SLF, 10,000 simulations.

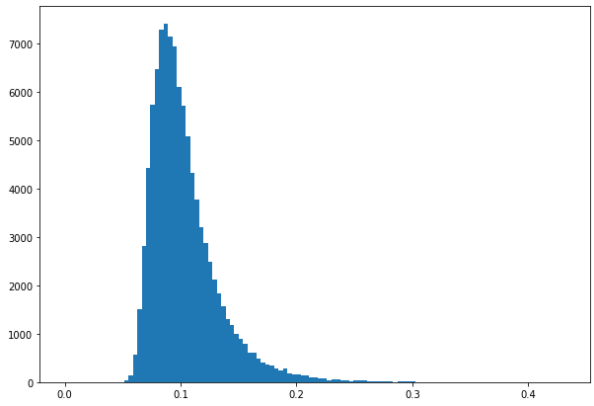

Figure 2.6: SLF, 100,000 simulations.

Figure 2.5 depicts F_n data from $10,000$ simulations. The estimated p-value produced by this simulation was 0.0043. In other words, our Monte Carlo simulation found that only 43 out of 10,000 simulations produced F_n values as extreme as that of the SLF data. Figure 2.6 depicts F_n data from $100,000$ simulations. The estimated p-value produced by this simulation was 0.00431, or $\frac{431}{100,000}$ F_n values produced from the allelic partitions of these Hoppe's Urn simulations are as extreme/large as the F_n value produced from the SLF data. These small p-values are large than those of the Coyne data simulations above, but are still

small enough that they are indicative of the SLF data not fitting the null-hypothesis of neutral evolution.

2.5.3 Simulation with Y chromosome Genetic Data from Underhill et al. (1997)

Since "the infinite alleles model is also relevant to DNA sequence data where there is no recombination" (such as with Y chromosomes) [D08, page 15], we'll look at Underhill et al.'s study. They found 20 distinct haplotypes in 718 Y chromosomes that they studied, with the following allelic partition:

$$a_1 = 7,\ a_2 = a_3 = a_5 = a_6 = a_8 = a_9 = a_{26} = a_{36} = a_{37} = 1,\ a_{82} = 2,\ a_{149} = 1,\ a_{266} = 1.$$

We created a instance of the Study class for data from Underhill's study of Y chromosomes. Durrett does not cite statistics from a similar process as ours, nor does he mention the Underhill data at all in the section of his book "Testing the infinite alleles model" [D08, page 25-26] – thus, it seems that simulations were not run for the Underhill data.

We chose to run our code for $N = 10,000$ and $N = 100,000$ successful infinite-allele simulations. Furthermore, Durrett presents 0.2133 as the F_n value of the Underhill data, and we used this F_n value for consistency with Durrett's simulations.

Our calculation of the F_n statistic (based on the allelic partition of Underhill's data which Durrett presents) found an F_n value of $27493/128881 = 0.213320815...$, so we input an F_n value of 0.2133.

As before, below are two histograms, which depict F_n value data in the same way as described with regard to the histograms in sections 3.5.1 and 3.5.2.

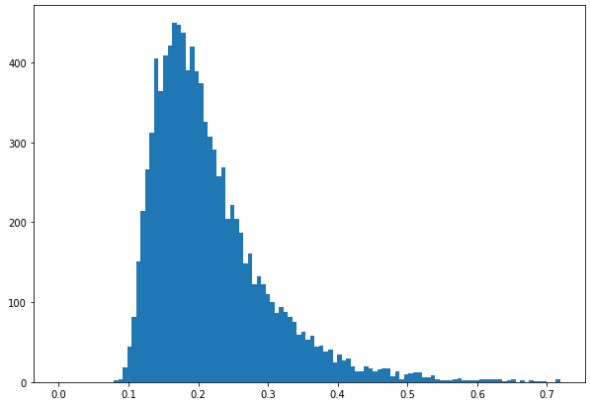

Figure 2.7: Underhill, 10,000 simulations.

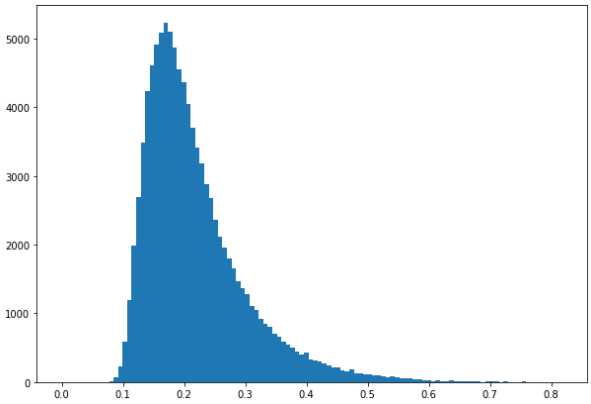

Figure 2.8: Underhill, 100,000 simulations.

Figure 2.7 depicts F_n data from $10,000$ simulations. The estimated p-value produced by this simulation was 0.409 ($\frac{4090}{10,000}$ simulations produced an F_n value as extreme/high as that

of the Underhill data). Figure 2.8 depicts F_n data from $100,000$ simulations. The estimated p-value produced by this simulation was 0.41355 ($\frac{41355}{10,000}$ simulations produced an F_n value as extreme/high as that of the Underhill data). Thus, the probability of obtaining an F_n value as extreme/large as the F_n statistic from the Underhill data is rather high – indeed, almost $\frac{1}{2}$. At this level probability, the null-hypothesis of neutral evolution (for the alleles studied in Underhill's paper) is not something which we would be inclined to reject.

3 Infinite Sites Model

3.1 Model Description

As DNA sequence data became more available, Kimura's infinite sites model – "in which mutations occur at distinct sites" – became more popular [D08, page 29-30]. In the infinite sites model we have one tree which branches, but branching (i.e. the addition of a lineage) is independent of new mutations arising. The amount of time that j lineages exist is denoted by t_j, and t_j "has approximately an exponential distribution" [D08, page 31]. The end of any t_j interval is marked by a branching event (not by a new mutation). The location of branching – that is, which lineage will now be split – follows the uniform distribution UNI(0,1). The location of the new mutation, i.e. which branch it occurs at, follows the uniform distribution UNI(0,1) – and the temporal distribution is exponential.

Figure 3.1 contains an illustrated example of a tree produced by the infinite sites model (the figure also includes the caption from Song's lecture notes):

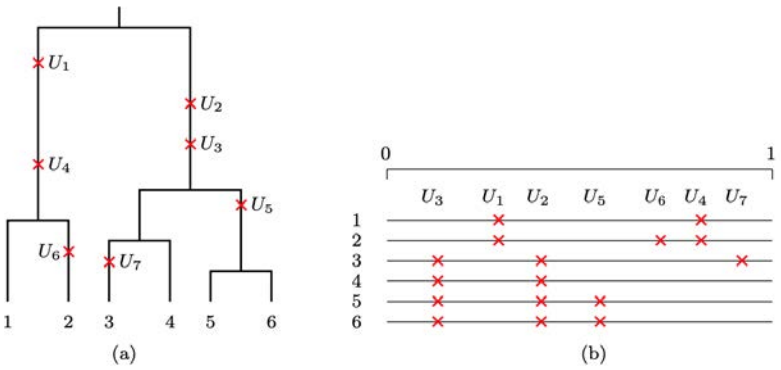

Fig. 5.1: Illustration of generating a sample under the infinite-sites model of mutation. (a) At each mutation event, a random number U_i is drawn from Unif$[0, 1]$. (b) Each leaf is assigned the unit interval $[0, 1]$ marked by the mutations encountered in the path from the leaf to the root of the tree.

Figure 3.1: Example of an Infinite Sites Tree [S21, page 68].

The infinite sites model can be used to address issues regarding segregating sites (sites where mutations occur). An example of one such question may be: what is the expected number of segregating sites in a sample size of n [D08, page 31]. However, questions regarding the age of a mutation may be addressed by this model [S21, page 73]. We focused our

simulations on this latter type of issue. Specifically, we designed and ran a Monte Carlo simulation with the aim of calculating the probability that a segregating site with b_1 descendants is older than another segregating site with b_2 descendants.

3.2 Computer Simulations

3.2.1 Node Objects

In our code, we defined a class of objects called "Node." Each Node object is really a node (where a branching of the tree occurs) and a particular branch that extends from that event. Similarly, each node has the potential for two descendants (a left one, and a right one), which arise when a branching occurs at the end of this current branch/node.

Each Node object also has a list of mutations, in chronological order of their occurrence on the branch. A Node also has a record of which other Node is its "parent," and has an age attribute (which is updated over time).

3.2.2 Function "build_tree()"

1 – Setting Up:

The function "build_tree()" takes in two inputs: n (the sample size, which is the size the population of the simulation should be at the end of the simulation) and θ (the statistic we discussed previously, which is related to the mutation rate).

In the function, we begin by defining the root Node/branch and two other Nodes (which we call "left" and "right") – which are also the first two descendants of the root Node, from after the first branching event. We call functions to set these Nodes, "left" and "right," as the left-descendant and right-descendant (respectively) of the root Node. And then we include these two newer-nodes in a list of active-Nodes, which lists which Nodes have branches that are still growing (the current-branch hasn't hit a branching event yet).

So far, everything has been set-up for the simulation. It is at this point that we set the absolute-time of our simulation to zero, and we begin moving the simulation forwards in time.

2 – Main While-Loop:

We enter a while-loop, in which we set a time (away from the current-time) for the next branching-event to occur. We do so by use of an exponential random variable, with a scale parameter – $\frac{1}{\lambda}$, where λ is the rate parameter – of $\frac{1}{\text{number of active Nodes}}$. Until we reach this time for the next branching event, we run along each of the active-branches and set times – using exponential random variables with $\frac{1}{\theta}$ as the scale-parameter (since θ is the statistic related

to the rate-parameter which we use, and the scale parameter is $\frac{1}{\text{rate-parameter}}$) – for mutations to happen. A mutation will occur on that branch if the time set for a mutation (on this particular branch) occurs before the time for the next branching event (which happens on one of the active-branches).

Now that we have reached the time for the next branching event, we update the absolute-time of the simulation. We also check how many active Nodes exist. If we have n active-Nodes, then we end the simulation – since we are interested in a tree with a certain number (n) of branches at the end. If fewer than n branches exist, we continue the simulation with the upcoming branching event.

We use a random integer generator to pick which of the k active-nodes will be split in this branching event – this ensures that the location of the mutation follows the uniform distribution. We use the function pop() to remove the Node in question from our list of active-nodes, and create two new Nodes ("new_left" and "new_right"). These Nodes age attributes are input in terms of absolute-time, and the Node which is being split (i.e. the Node-in-question which was just popped from the active-nodes list) has it's left-descendant and right-descendant attributes set with these new Nodes. And finally, we append these two new Nodes to the active-nodes list.

3 – Summing Up:

We described above how after an initial setting up part of the code (in this function) we are dealing with code occurring inside of a while-loop, and that there is a check for whether there are n-active nodes (in order to determine whether to stop the simulation). Thus, now that we have described the algorithm inside the while-loop, we understand that the tree continues to grow and branch, with mutations occurring along the way, until we end up with n leaves of the tree – n active Nodes – and having reached our desired n, we have grown the tree to completion.

The function returns the root Node, the active-nodes (or "leaves" of the tree), the length of the simulation (i.e. absolute-time at the end of the simulation), and a dictionary with the mutations (or, to be exact, the mutation-times) from the simulation as keys.

3.2.3 Function "older_MC_sim()"

This function, older_MC_sim(), runs a Monte Carlo simulation with the aim of calculating the probability that a segregating site with b_1 descendants is older than another segregating site with b_2 descendants. The inputs are n (the sample population size), θ (a statistic related to the mutation rate), b_1, b_2, and N (the number of infinite sites simulations which will be run for this Monte Carlo simulation). We have the constraints that $0 < b1 \leq b2$. In other words, this function aims to calculate the probability that a segregating site with $b_1 \neq 0$ descendants is older than another segregating site with a greater or equal number ($b_2 \neq 0$) of descendants. We should note however, that the case of $b_1 = b_2$ is a trivial case

($\mathbb{P} = \frac{1}{2}$). As this is the case, we did not design our code to deal with the trivial case of $b_1 = b_2$.

We make a while-loop, which will continue until N simulations, in which there is a mutation with b_1 descendants and another mutation with b_2 descendants, are completed. Inside the loop, we call build_tree() to run an infinite sites simulation. We then correlate the mutations (returned, as times that they occurred, in a dictionary) with their descendants, using the following methodology: For each leaf/final-branch of the tree, we look at each the mutation-times on that branch and on all of its ancestor-branches. For each of those mutation-times, we enter the dictionary (with mutation-times as keys) and add the current leaf to a list (which is the value corresponding to this mutation-time/key). Thus, we end up with a dictionary with mutation-times as keys – for which a list of leaves which contain the mutation-in-question is the value corresponding to the key.

We then convert the dictionary into a list, and randomly shuffle the list, before finally converting the list into an ordered-dictionary. In general terms, we do this so that the mutations are not ordered in any more-or-less chronological way (as the mutations were input into the dictionary originally) that would bias the analysis (which is concerned with probability connected with the chronology of mutations).

We now will analyze the data from our infinite sites simulation. We iterate through the mutations-dictionary to check to see if there is a mutation with b_1 descendants. If there is, we select that mutation, that entry of the dictionary (let's call it "mutation-b1" for now). Then we iterate through the dictionary again, and look for a mutation with b_2 descendants – and if it exists, we'll select that entry too (and term it "mutation-b2"). Now that we have a simulation which produced two such mutations, we make note of it by incrementing a counter (called "counter"). We then compare the entries to see which mutation is older (which is easy, since every dictionary-entry's key is the time that the mutation-in-question occurred). If mutation-b1 is older than mutation-b2, we increment another counter (called "success").

Once "counter" has reached N (that is, N many simulations with a mutation with b_1 descendants and another mutation with b_2 descendants were run and analyzed), we calculate the probability (observed through this Monte Carlo simulation) that the mutation-site with b_1 descendants is older than one with b_2 descendants. The calculation is the following: $\frac{success}{N}$ – that is, the number of times we observed that the mutation-site with b_1 descendants is older than one with b_2 descendants, over the total number of simulations which produced a mutation-site with b_1 descendants and another with b_2 descendants.

3.2.4 Results

We called the older_MC_sim() function for "data from Ward et al. (1991), who sequenced 360 nucleotides in the D loop of mitochondria in 63 humans" [D08, page 30]. We had $n = 63$, we chose $\theta = 1$, and $N = 1000$. We focused on the mutations at positions 296 and 302 in the sequenced genetic fragment. 28 members of the sample population carried the mutation at site 296, and 24 members carried the mutation at site 302. Thus we have $b_1 = 24$ and $b_2 = 28$.

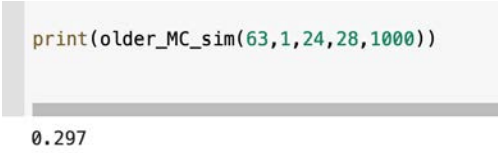

Figure 3.2: Estimated Probability Monte Carlo Simulation for Ward et al. Data.

As we see in Figure 3.2, the estimated probability (based on a Monte Carlo simulation) that the mutation with $b_1 = 24$ descendants (in our sample) is older than the mutation with $b_2 = 28$ descendants is 0.297.

4 Appendices

4.1 Appendix 1 – Infinite Alleles Monte Carlo Python Simulation

Link to view our Google Colab file ("Hoppe urn.ipynb"):
`https://colab.research.google.com/drive/18KUeuJMw0wJBXKnNUaiHa25gVI9murop?usp=sharing`.

Below is the text of our code:

```
from enum import Enum, auto
import math
import random
import plotly.express as px
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
```

```
# pValue is the p-value, which is the probability of obtaining test results
# at least as extreme as the result actually observed (under the assumption
# that the null hypothesis is correct). In our case, it is the probability of
# producing an allelic partition (with a Hoppe's Urn simulation, which assumes
# neutral theory of molecular evolution for this gene locus - that is, this
# evolution of this gene locus is not based on fitness of the mutations,
# rather it is a product of stocastic processes) that is as "extreme" (probably
# meaning: as unevenly partitioned) as an allelic partition observed in nature.
```

```
class Study:
  def __init__(self,numOfSimulations,n,K_n,StudyFn):
    self.numOfSimulations = numOfSimulations #Number of simulations with K_n
                                             #different genes produced.
    self.populationSize = n                  #Population size.
    self.numOfDiffGenes = K_n                #Number of different alleles (or
                                             #haplotypes) at the gene locus in
                                             #question.
    self.StudyFn = StudyFn                   #F_n statistic, calculated from the
                                             #partition in the study.
    self.theta,self.pValue,self.fVals = infiniteAlleles(numOfSimulations,n,K_n,StudyFn)
```

```
# A function which simulates Hoppe's Urn, with inputs n and theta.

def generate(n = 60, theta = 5):
  num_of_trees = 1 #Start out with 1 tree - the mutation.
  tree_sizes = []
  for i in range(n):
    x = (i+theta) * random.random()
    cumsum = 0
    old_color = False
    for j,value in enumerate(tree_sizes):
      cumsum += value
      if x < cumsum:
        tree_sizes[j] += 1
        old_color = True
        break
    if not old_color:
      tree_sizes.append(1)
      num_of_trees += 1

  return (num_of_trees,tree_sizes)
```

```
# F_n "is the probability that two individuals chosen with replacement are the
# same" (Durrett's textbook, pp. 25-26). Inputs are n (population size) and a
# dictionary with j's (i.e. number of representatives) as the keys and a_j's as
# the value-pairs (i.e. how many alleles have this number of representatives).

def Fn(n, thisdict): #F_n = sum[a_j * (j/n)^2] for j=1 to n.
```

```
  sum = 0
  for x in thisdict:
    j = x
    aj = thisdict[x]
    sum += aj * (j/n)**2
  return sum



# Theta^hat is a sufficient statistic for theta = 4N*mu (where mu is the
# mutation rate). Theta is related to the mutation rate,
# and is used to calculate the probabilities of both a coalescence event
# and a mutation event.
# Theta^hat = K_n / log(n), where K_n is the number of different alleles,
# and n is the population size.

def findTheta(n,kn):
  theta = kn/math.log(n)
  return theta



# Run infinite alleles model. Inputs: N, number of simulations; n, population size;
# K_n, number of different alleles/haplotypes; F_n value, calculated from the
# partition in the study.

def infiniteAlleles(N,n,kn,FVal):
  theta = findTheta(n,kn)
  sample_counter = 0         # Counts number of simulations with K_n as the number
                             # of different alleles (i.e. successful simulations).
  larger_counter = 0         # Counts number of simulations with an F_n which is
                             # greater than the F_n-value of the study.
  fVals = []                 # List that records F_n values of successful simulations.
  while sample_counter < N:  # While-loop that continues until there are N-number
                             # of successful simulations.
    new_sample = generate(n,theta)                  #Run a Hoppe's Urn simulation.
    if new_sample[0] == kn:                         #Select simulations that have K_n
                                                    #as the number of different alleles.
      #print(new_sample[1])
      sample_counter += 1
      MyList = new_sample[1]                        #List of sizes of the trees
                                                    #(i.e. how many representatives
                                                    #each allele has; allele has j
```

```
                                                    #representatives)
      my_dict = {i:MyList.count(i) for i in MyList} #Make a dictionary that counts
                                                    #how many alleles (trees) have
                                                    #the same number of representatives
                                                    #(descendants); calculate the
                                                    #a_j's. Key:value-pairs, j:a_j.
      fVal = Fn(n,my_dict)                          #Use j's and a_j's to calculate F_n.
      fVals.append(fVal)                            #Add to a vector of F_n values.
      if fVal > FVal:                               #Count how many times this
                                                    #simulated-partition's F_n is
                                                    #greater than the F_n value of
                                                    #the study.
        larger_counter += 1

  return theta, larger_counter/N, fVals



# Coyne (1976), Drosophila persimilis.
FVal = 0.2972
Coyne = Study(100000,60,23,FVal)       #N=1000 from book;
                                       #FVal from book, self calculation
                                       #produced 107/360=0.2972222... .

print(Coyne.pValue)

# Creating an array of F_n values from the successful simulations.
a = np.array(Coyne.fVals)

# Creating histogram with the array of F_n values.
fig, ax = plt.subplots(figsize =(10, 7))
ax.hist(a, bins = 100)
ax.plot(FVal)

# Show plot.
plt.show()



#Singh, Lewontin, and Felton (1976), D. pseudoobscura.
FVal = 0.2353
SLF = Study(100000,146,27,FVal)        #N=2000 from book;
                                       #the FVal inputed from book, but self calculation
```

```
                                        produced 2513/10658=0.235785325577031337..... .

print(SLF.pValue)

# Creating an array of F_n values from the successful simulations.
a = np.array(SLF.fVals)

# Creating histogram with the array of F_n values.
fig, ax = plt.subplots(figsize =(10, 7))
ax.hist(a, bins = 100)
ax.plot(FVal)

# Show plot.
plt.show()



#Underhill et al. (1997), Y chromosomes -- 20 distinct haplotypes.
FVal = 0.2133
Underhill = Study(100000,718,20,FVal)  #N up to us;
                                       #FVal self calculation produced
                                       #27493/128881=0.213320815..... .

print(Underhill.pValue)

# Creating an array of F_n values from the successful simulations.
a = np.array(Underhill.fVals)

# Creating histogram with the array of F_n values.
fig, ax = plt.subplots(figsize =(10, 7))
ax.hist(a, bins = 100)
ax.plot(FVal)

# Show plot.
plt.show()
```

4.2 Appendix 2 – Infinite Sites Monte Carlo Python Simulation

Link to view our Google Colab file ("Infinite site model MC simulation.ipynb"):
https://colab.research.google.com/drive/1ZstsyDw2VnXKa7adSsJzNuDSgqGGOQVf?usp=sharing.

Below is the text of our code:

```
from enum import Enum, auto
import math
import random
import collections
import plotly.express as px
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt


class Node:
  def __init__(self, parent=None): #parentID=None, ID=None
    self.left = None
    self.right = None
    self.mutations = []
    self.parent = parent
    self.age = 0
    self.ID = None
    self.parentID = None
  def set_left(self, x):
    self.left = x
  def set_right(self, x):
    self.right = x
  def add_mutation(self, x):
    self.mutations.append(x)
  def add_age(self, x):
    self.age += x



def build_tree(n, theta):
  mutations = {}
  root = Node()
  left = Node(root)  #parent is root
  right = Node(root) #parent is root
  root.set_left(left) #left-branch of old-node is left-new-node
  root.set_right(right) #right-branch of old-node is right-new-node
  active_nodes = [left, right]

  absolute_time = 0
  while True:
    split_time = np.random.exponential(1/len(active_nodes)) #next un-coalescent
                                                            #event is calculated with
                                                            #r.v. exp(1/lambda),
```

```
                                                        #where lambda=rate,
                                                        #1/lambda is the scale
                                                        #parameter.
    for node in active_nodes:
      #Add mutation events
      current_time = 0
      while True:
        current_time += np.random.exponential(1/theta)
        if current_time < split_time:
          node.add_mutation(node.age + current_time)
          mutations[absolute_time + current_time] = []
        else:
          break
      #Update node age
      node.add_age(split_time)

    absolute_time += split_time

    if len(active_nodes) == n:
      break

    #Split one node
    split_index = random.randint(0,len(active_nodes)-1) #pick node to split;
                                                        #random number between
                                                        #0 and k-1 (both included)
    split_node = active_nodes.pop(split_index)
    new_left = Node(split_node)
    new_right = Node(split_node)
    new_left.add_age(absolute_time)
    new_right.add_age(absolute_time)
    split_node.set_left(new_left)
    split_node.set_right(new_right)
    active_nodes = active_nodes + [new_left, new_right]

  return (root,active_nodes,absolute_time,mutations)




def older_MC_sim(n,theta,b1,b2,N):
  #Simulates whether segregating site with b1 descendants is older than another one
  #    with b2 descendants
  #N: number of simlations
  #0 < b1 < b2
```

```
counter = 0
success = 0
while True:
  (root,leaves,abs_time,mutations) = build_tree(n,theta)
  for leafe in leaves:
    curr = leafe
    while curr != root:
      for mutation in curr.mutations:
        mutations[mutation].append(leafe)
      curr = curr.parent

  items = list(mutations.items())
  random.shuffle(items)
  mutations = collections.OrderedDict(items)

  tree_used = False
  for site1 in mutations:
    if tree_used:
      break
    if len(mutations[site1]) == b1:
      for site2 in mutations:
        if len(mutations[site2]) == b2: #need b2 not equal b1 - because if it is,
                                        #the code would probably pick the same
                                        #mutation for both selections.
          counter += 1
          if site1 < site2: #site1 < site2 means the value of site2 is bigger,
                            #i.e. further along in time, and therefore site2
                            #is younger.
            success += 1
          if counter == N:
            return success / N
          tree_used = True
          break
```

References

[B23] Beers, B. "P-Value: What It Is, How to Calculate It, and Why It Matters," *Investopedia*, 28 Mar. 2023. https://www.investopedia.com/terms/p/p-value.asp.

[C76] Coyne, J.A. "Lack of genic similarity between two sibling species of *Drosophila* as revealed by varied techniques," *Genetics*, 84, pp. 593–607, 1976.

[D08] Durrett, R. "Probability Models for DNA Sequence Evolution," 2nd edition, 2008. https://services.math.duke.edu/~rtd/Gbook/PM4DNA_0317.pdf.

[JT87] Joyce, P., and Tavaré, S. "Cycles, permutations and the structure of the Yule process with immigration" *Stoch. Proc. Appl.*, 25, pp. 309–314, 1987.

[N18] Nordborg M. "Introduction to the coalescent theory - Lecture 1 by Magnus Nordborg," Youtube, *Internation Centre for Theoretical Studies*. Posted on June 12, 2018. Retrieved on April 28, 2023. https://www.youtube.com/watch?v=0j0jW0stbB8.

[SLF76] Singh, R.S., Lewontin, R.C., and Felton, A.A. "Genetic heterogeneity within electrophoretic "alleles" of xanthine dehydrogenase in *Drosophila pseudoobscura*," *Genetics*, 84, pp. 609–629, 1976.

[S21] Song Y.S. "Lecture Notes on Computational and Mathematical Population Genetics," 2021. https://people.eecs.berkeley.edu/~yss/Pub/CMPG_lecture_notes.pdf.

[U97] Underhill, P.A. et al. "Detection of numerous Y chromosome bial- lelic polymorhpisms by denaturing high performance liquid chromatography," *Genome Research*, 7, pp. 996–1005, 1997.

[W91] Ward, R.H., Frazier, B.L., Dew-Jager, K., and Pääbo, S. "Extensive mitochondrial diversity within a single Amerindian tribe," *Proc. Natl. Acad. Sci. USA.*, 88, pp. 8720–8724, 1991.

[W75] Watterson, G.A. "On the number of segregating sites in genetical models without recombination," *Theor. Pop. Biol.*, 7, pp. 256–276, 1975.

[W77] Watterson, G.A. "Heterosis or neutrality?" *Genetics*, 85, pp. 789–814, 1977.