

Gene genealogies in haploid populations evolving according to sweepstakes reproduction — approximating $\mathbb{E} \left[\tilde{R}_i^N(n) \right]$

BJARKI ELTON^{1 2} 

Consider a haploid panmictic population of constant size N evolving according to some given model. Suppose we *(i)* evolve the population forward in time for an infinite amount of time and record the ancestral relations of everyone in the population at any time; *(ii)* at some given time we record a random sample from the population. Since we know the entire population ancestry the tree relating the sampled individuals is fixed; *(iii)* trace the sample tree and record the branch lengths of the fixed sample tree. After erasing the population ancestry and the information about the sampled individuals we repeat steps *(i)*–*(iii)* a given number of times, each time starting from scratch with a new population ancestry independent of any previous (and future) ancestries. We denote the mean relative branch lengths obtained in this way with $\mathbb{E} \left[\tilde{R}_i^N(n) \right]$. With this C++ simulation code we approximate $\mathbb{E} \left[\tilde{R}_i^N(n) \right]$

Contents

1	intro	2
2	Copyright	4
3	compilation and output	5
4	code	6
4.1	the included libraries	7
4.2	the random number generators	8
4.3	probability mass function for number of offspring	9
4.4	generate cumulative mass function	10
4.5	sample one bounded random number of potential offspring	11
4.6	one unbounded random number of potential offspring	12
4.7	add one generation to population ancestry	13
4.8	add one generation to population ancestry when Model 1	14
4.9	get a random sample	15
4.10	look up the immediate ancestor	16
4.11	check if sample tree is complete	17
4.12	sort in descending order	18
4.13	update sample	19
4.14	update $\ell_i^N(n)$	21
4.15	update $r_i^N(n)$	22
4.16	one experiment	23
4.17	approximate $\mathbb{E} \left[\tilde{R}_i^N(n) \right]$	24
4.18	the main module	25

¹beldon11@gmail.com

²compiled @ 12:01pm on Wednesday 22nd October, 2025

CTANGLE 4.12.1 (TeX Live 2025/Debian)

g++ (Debian 15.2.0-4) 15.2.0

6.16.12+deb14-amd64 GNU/Linux

GNU bash, version 5.3.3(1)-release (x86_64-pc-linux-gnu)

GSL 2.8

CWEAVE 4.12.1 (TeX Live 2025/Debian)

X_YLaTeX XeTeX 3.141592653-2.6-0.999997 (TeX Live 2025/Debian)

GNU parallel 20240222

GNU Awk 5.3.2, API 4.0, PMA Avon 8-g1, (GNU MPFR 4.2.2, GNU MP 6.3.0)

SpiX 1.3.0

GNU Emacs 30.1

1 intro

Gene genealogies are generated through inheritance as the population evolves forward in time. This means that the individuals (gene copies) in a given sample are related through one fixed tree. Coalescents (Markov processes on partitions of positive integers describing the random ancestral relations of sampled gene copies) are derived by describing the ancestral relations of a sample one generation at a time going backwards in time from the sample, thus ignoring the fact that the sampled gene copies are related through a fixed complete tree.

Here we provide the means to investigate functionals of fixed complete trees. An effort to illustrate the idea may be found in Figure 1. Suppose a haploid panmictic population of constant size has evolved according to a given model for an infinite number of generations (Figure 1(i)). The ancestry of all the individuals in the population at any time is known and is stored in a sigma field. At time 0 we record a random sample (the \bullet in Figure 1(ii)). Since the ancestry of the population is known arbitrarily far back in time, the sampled gene copies will have a fixed complete tree given the population ancestry (marked out by the \bullet in Figure 1(iii)).

Write $[n] \equiv \{1, 2, \dots, n\}$ for any $n \in \mathbb{N} \equiv \{1, 2, \dots\}$. Let $\tilde{L}_i^N(n)$ denote the random length of branches supporting i leaves (sampled gene copies) for all $i \in [n-1]$ read off a fixed tree given a random population ancestry. Write $\tilde{R}_i^N(n) \equiv \tilde{L}_i^N(n) / \sum_j \tilde{L}_j^N(n)$. We are interested in approximating $\mathbb{E}[\tilde{R}_i^N(n)]$ by averaging realisations of $\tilde{R}_i^N(n)$ over independent ancestries (Figure 2). In Figure 2 we repeat the process in Figure 1 some given number (M) of times, each time reading branch lengths off fixed complete trees. For example, for the gene genealogy in Figure 4 we have $\tilde{L}_1^N(3) = 4$, and $\tilde{L}_2^N(3) = 1$.

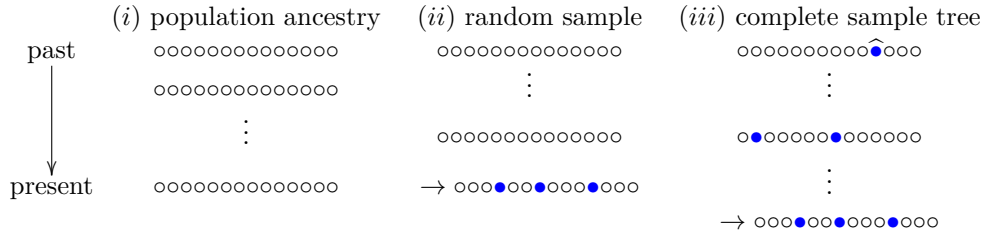


Figure 1: A fixed complete sample tree is obtained in the three steps shown; the most recent common ancestor of the sampled gene copies (\bullet) is marked $\hat{\bullet}$

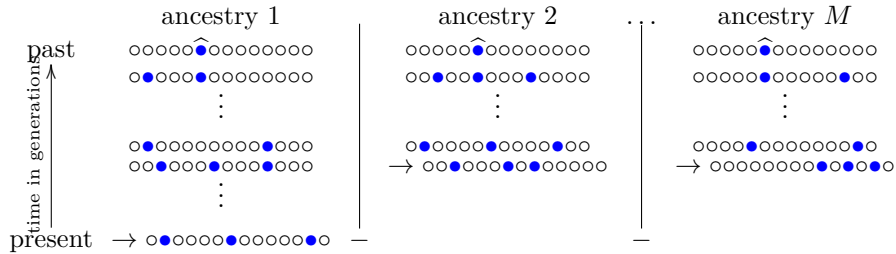


Figure 2: Complete sample trees within independent population ancestries

At any given time the current individuals independently produce potential offspring according

to a given law. For all $k \in [\zeta(N)]$ suppose

$$\mathbb{P}(X = k) = C(k^{-\alpha} - (1 + k)^{-\alpha}) \quad (1)$$

where $C > 0$ is chosen such that $\mathbb{P}(X \in [\zeta(N)]) = 1$. The quantity $\zeta(N) \in \mathbb{N}$ is an upper bound on the number of potential offspring an individual may produce. From the pool of potential offspring we sample (uniformly without replacement) a fixed number to survive. Let $(\varepsilon_N)_N$ be a sequence of positive constants where $0 < \varepsilon_N < 1$ and it may hold that $\varepsilon_N \rightarrow 0$ as $N \rightarrow \infty$. Write

$$X \triangleright L(a, \zeta(N))$$

when X is distributed according to (1) with a and $\zeta(N)$ as given.

We will consider two models for the law on the number of potential offspring, ‘Model 0’ and ‘Model 1’. We now define Model 0. Let E be the event

$$\{X_i \triangleright L(\alpha, \zeta(N)) \text{ for all } i \in [N]\} \quad (2)$$

and E^c the event where κ replaces α in E in (2). Suppose $\mathbb{P}(E) = \varepsilon_N$ and $\mathbb{P}(E^c) = 1 - \varepsilon_N$. We call this ‘Model 0’ (**MODEL** = 0).

We now define Model 1. Let E_1 be the event

$$\begin{aligned} &\{\text{there exists exactly one } i \in [N] \text{ with } X_i \triangleright L(\alpha, \zeta(N)); \\ &\text{moreover, it holds that } X_j \triangleright L(\kappa, \zeta(N)) \text{ for all } j \in [N] \setminus \{i\}\} \end{aligned} \quad (3)$$

Let E_1^c be the event where κ replaces α in E_1 ; when E_1^c occurs it holds that $X_i \triangleright L(\kappa, \zeta(N))$ for all $i \in [N]$. Suppose $\mathbb{P}(E_1) = \bar{\varepsilon}_N$ and $\mathbb{P}(E_1^c) = 1 - \bar{\varepsilon}_N$ and it may hold that $\bar{\varepsilon}_N \rightarrow 0$ as $N \rightarrow \infty$. This is ‘Model 1’ (**MODEL** = 1).

We record the ancestry, i.e. the immediate ancestor of every individual in the population at any time. Thus, as the population evolves forward in time we generate gene genealogies. Given a random sample, the sampled gene copies share a fixed complete tree.

In § 4 we summarise the algorithm, the code follows in § 4.1 – § 4.18, we conclude in § 5. Comments within the code are in **this font and colour**

2 Copyright

Copyright © 2025 Bjarki Eldon

This document and any source code it contains is distributed under the terms of the GNU General Public Licence (version ≥ 3). You should have received a copy of the licence along with this file (see file COPYING).

The source codes described in this document are free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This document and the code it contains is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this file (see COPYING). If not, see <http://www.gnu.org/licenses/>.

3 compilation and output

This CWEB [KL94] document (the `.w` file) can be compiled with `cweave` to generate a `.tex` file, and with `ctangle` to generate a `.c` [KR88] C++ code file.

To produce this document one can use the shell tool `spix` on the source file

```
spix <prefix>.w
```

thus running the shell script appearing before the preamble in the source file (`.w` file), the lines beginning with `%%`

One may also copy the shell script (minus the leading `%%`) into a file (say, `scriptfile`) and use `parallel` [Tan11]

```
parallel --gnu -j1 ::: ./scriptfile
```

Use `valgrind` to check for memory leaks:

```
valgrind -v --leak-check=full --show-leak-kinds=all <program call>
```

Use `cppcheck` to check the code

```
cppcheck --enable=all --language=c++ <prefix>.c
```

Figure 5 holds an example output

4 code

we briefly summarise the main steps of the algorithm

1. $(r_1^N(n), \dots, r_{n-1}^N(n)) \leftarrow (0, \dots, 0)$
2. for each of M experiments § 4.17
 - (a) $(\ell_1^N(n), \dots, \ell_{n-1}^N(n)) \leftarrow (0, \dots, 0)$
 - (b) initialise population ancestry $a_i(0) = i$ for $i = 0, 1, \dots, N-1$
 - (c) sample (uniformly without replacement) random number n of levels § 4.9
 - (d) **while** sample tree is not complete: § 4.11
 - i. add one generation to the population ancestry § 4.7
 - ii. take a new random sample § 4.9
 - (e) read lengths $\ell_i^N(n)$ off the fixed complete sample tree § 4.16
 - (f) given the lengths $\ell_i^N(n)$ update the approximations $r_i^N(n)$ § 4.15
3. return $r_i^N(n)/M$ as an approximation of $\mathbb{E} \left[\tilde{R}_i^N(n) \right]$ for $i = 1, 2, \dots, n-1$

The population evolves as a supercritical branching process conditioned to be of a fixed size. We record the ancestry of the individuals in the population at any time. The individuals at any time are seen as occupying levels, one individual on each of N levels. The individual on level i at time g has immediate ancestor $a_i(g)$, where $a_i(g)$ is the level of the parent of the given individual. The offspring of the individual on level i will all have value i ; each offspring ‘points’ to its’ immediate ancestor (Figure 3) Gene genealogies evolve forward in time (Figure 4). The

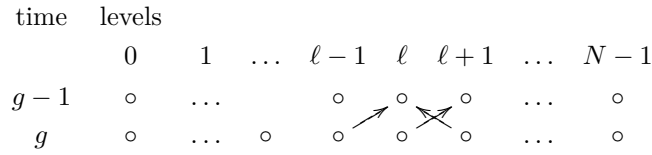


Figure 3: The immediate ancestor of the individuals on levels $\ell-1$ and $\ell+1$ at time g occupies level ℓ at time $g-1$ ($a_{\ell-1}(g) = a_{\ell+1}(g) = \ell$); and $a_\ell(g) = \ell+1$

individuals (gene copies) of any given sample will share one complete gene genealogy. Given that we know the population ancestry we can use this information to read gene genealogical statistics off fixed complete sample trees.

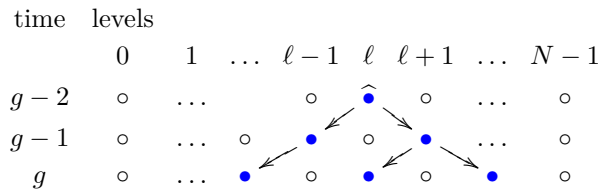


Figure 4: The gene genealogy of the 3 individuals sampled at time g

4.1 the included libraries

the included libraries

```
5  ⟨included libraries 5⟩ ≡  
    #include <iostream>  
    #include <fstream>  
    #include <vector>  
    #include <random>  
    #include <functional>  
    #include <memory>  
    #include <utility>  
    #include <algorithm>  
    #include <ctime>  
    #include <cstdlib>  
    #include <cmath>  
    #include <list>  
    #include <string>  
    #include <fstream>  
    #include <chrono>  
    #include <forward_list>  
    #include <assert.h>  
    #include <math.h>  
    #include <unistd.h>  
    #include <gsl/gsl_rng.h>  
    #include <gsl/gsl_randist.h>  
    #include <gsl/gsl_math.h>  
    #include "quenched_mixed_bounded_sfswplotting.hpp"
```

This code is used in chunk 22.

4.2 the random number generators

define the random number generators

```
6  <random number generators 6> ≡      /*
    obtain a seed out of thin air for the random number engine */
    std::random_device randomseed;      /*
    Standard mersenne twister random number engine seeded with rng() */
    std::mt19937_64 rng(randomseed());
    gsl_rng * rngtype;
    void setup_rng(unsigned long int s)
    {
        const gsl_rng_type *T;
        gsl_rng_env_setup();
        T = gsl_rng_default;
        rngtype = gsl_rng_alloc(T);
        gsl_rng_set(rngtype, s);
    }
```

This code is used in chunk 22.

4.3 probability mass function for number of offspring

The mass function (1)

7 \langle mass function 7 $\rangle \equiv$

```
double kernel ( const std ::size_t k, const double a )
{
    /*
    a is either  $\alpha$  ALPHA or  $\kappa$  KAPPA */
    return (pow(1./static_cast<double>(k), a) - pow(1./static_cast<double>(k + 1),
    a));
}
```

This code is used in chunk 22.

4.4 generate cumulative mass function

generate the cumulative mass function for sampling numbers of potential offspring

8 $\langle \text{cmf for number of offspring } 8 \rangle \equiv$

```
void generate_cmfs ( std::vector < double > &cmfa, std::vector < double > &cmfA ) {  
  double s  
  { }  
  ;  
  double d  
  { }  
  ;  
  for ( std::size_t i = 1; i ≤ CUTOFF; ++i ) {  
    cmfa[i] = cmfa[i - 1] + kernel(i, ALPHA);  
    cmfA[i] = cmfA[i - 1] + kernel(i, KAPPA);  
    s += kernel(i, ALPHA);  
    d += kernel(i, KAPPA);  
  }  
  assert(s > 0.);  
  assert(d > 0);  
  std::transform ( cmfa.begin(), cmfa.end(), cmfa.begin(), [&s](const double &x)  
  {  
    return x/s;  
  }  
  ) ; std::transform ( cmfA.begin(), cmfA.end(), cmfA.begin(), [&d](const double &x)  
  {  
    return x/d;  
  }  
  ) ;  
  assert(abs(cmfa.back() - 1.) < 0.999999);  
  assert(abs(cmfA.back() - 1.) < 0.999999);  
  assert(abs(cmfa[0]) < 0.000001);  
  assert(abs(cmfA[0]) < 0.000001); }
```

This code is used in chunk 22.

4.5 sample one bounded random number of potential offspring

sample one bounded random number of potential offspring using (1)

9 \langle one bounded number 9 $\rangle \equiv$

```
static std::size_t randomX ( const std::vector< double > &f )
{
    /*
        rngtype § 4.2 */
    const double u = gsl_rng_uniform_pos(rngtype);
    std::size_t j
    {1};
    while (u > f[j]) {
        ++j;
        assert(j ≤ CUTOFF);
    }
    return j;
}
```

This code is used in chunk 22.

4.6 one unbounded random number of potential offspring

sample one unbounded random number of potential offspring as $\lfloor U^{-1/\alpha} \rfloor$ where U is a standard random uniform and $\alpha > 0$ is fixed

10 $\langle \text{an unbounded } X \text{ }_{10} \rangle \equiv$

```
static std::size_t unboundedrandomX()  
{  
    return static_cast<std::size_t>(floor(pow(1./gsl_rng_uniform_pos(rngtype),  
        1./ALPHA)));  
}
```

This code is used in chunk 22.

4.7 add one generation to population ancestry

add one generation to population ancestry; the offspring (if any) of the individual on level i will have value i (point to i). Recall X_1, \dots, X_N are the random numbers of potential offspring produced by the current N individuals, by (1) $X_1 + \dots + X_N \geq N$ almost surely. We add to the ancestry in the following 3 steps:

1. sample X_1, \dots, X_N and append indexes $(\dots, \underbrace{1, \dots, 1}_{X_1}, \dots, \underbrace{N, \dots, N}_{X_N})$
2. shuffle the new $X_1 + \dots + X_N$ indexes $(\dots, \sigma_1, \dots, \sigma_{X_1 + \dots + X_N})$
3. keep the first N of the new (shuffled) indexes $(\dots, \sigma_1, \dots, \sigma_N)$

11 $\langle \text{one generation 11} \rangle \equiv$

```
void addgeneration_modelnull (std::vector < std::size_t > &tree, const std::vector <
    double > &vcmf )
{
    const std
        ::size_t e = tree.size();    /*
        step 1 */
    std::size_t x
    {}
    ;
    for (std::size_t i = 0; i < POP_SIZE; ++i) {    /*
        randomX § 4.5 */
        x = randomX(vcmf);
        tree.reserve(tree.size() + x);    /*
        the x offspring of individual on level i point to i; recall Figure 4 */
        tree.insert(tree.end(), x, i);
    }
    assert(tree.size() ≥ e + POP_SIZE);    /*
    step 2 */
    std::shuffle(tree.begin() + e, tree.end(), rng);    /*
    step 3 */
    tree.resize(e + POP_SIZE);    /*
    population size N POP_SIZE is fixed so POP_SIZE must divide the size of the ancestry
    */
    assert((tree.size() % POP_SIZE) == 0);
}
```

This code is used in chunk 22.

4.8 add one generation to population ancestry when Model 1

add one generation to the population ancestry when Model 1 holds

12 \langle add a generation for model 1 12 $\rangle \equiv$

```

void addgeneration_modelone (std::vector < std::size_t > &tree, const std::vector <
    double > &vcmf_a, const std::vector < double > &vcmf_A )
{
    const std
        ::size_t e = tree.size();
    const double u = gsl_rng_uniform_pos(rngtype);
    std::size_t x = randomX(u < EPSILON ? vcmf_a : vcmf_A);
    for (std::size_t i = 1; i < POP_SIZE; ++i) {
        x = randomX(vcmf_A);
        tree.reserve(tree.size() + x);
        tree.insert(tree.end(), x, i);
    }
    assert(tree.size() ≥ e + POP_SIZE);
    std::shuffle(tree.begin() + e, tree.end(), rng);
    tree.resize(e + POP_SIZE);
    assert((tree.size() % POP_SIZE) ≡ 0);
}

```

This code is used in chunk 22.

4.9 get a random sample

get a random sample; return the levels of the sampled individuals (gene copies)

13 \langle random sample 13 $\rangle \equiv$

```
static void randomsample(std::vector< std::pair< std::size_t, std::size_t >> &sample)
{
    /*
        pair is (size of block, level of block) */
    std::vector< std::size_t > V(POP_SIZE, 0);
    std::iota(V.begin(), V.end(), 0);
    std::shuffle(V.begin(), V.end(), rng);
    sample.clear();
    assert(sample.size() < 1);
    sample.reserve(SAMPLE_SIZE);
    for (std::size_t i = 0; i < SAMPLE_SIZE; ++i) {
        /*
            pair is (size of block, level of block) */
        sample.push_back(std::make_pair(1, V[i]));
    }
    assert(sample.size() == SAMPLE_SIZE);
}
```

This code is used in chunk 22.

4.10 look up the immediate ancestor

look up the immediate ancestor; $a_i(g) \in \{0, 1, \dots, N-1\}$ is the level of the immediate ancestor of the individual on level i at time g . The population ancestry is the vector (N is population size)

$(\dots, a_0(g-1), a_0(g-1), \dots, a_{N-1}(g-1), \dots, a_0(g), \dots, a_{N-1}(g), a_0(g+1), \dots, a_{N-1}(g+1), \dots)$

where $a_i(0) = i$ for $i = 0, 1, \dots, N-1$;

time	level
0	$0, 1, \dots, N-1$
1	$a_0(1), a_1(1), \dots, a_{N-1}(1)$
\vdots	
$g-1$	$a_0(g-1), a_1(g-1), \dots, a_{N-1}(g-1)$
g	$a_0(g), a_1(g), \dots, a_{N-1}(g)$
$g+1$	$a_0(g+1), a_1(g+1), \dots, a_{N-1}(g+1)$
\vdots	

14 \langle get immediate ancestor 14 $\rangle \equiv$

```
static std::size_t getagi ( const std::size_t &g, const std::size_t &i, const std::vector <
                        std::size_t > &tree )
{
    /*
        return  $a_i(g) = \text{tree}[(gN) + i]$  , the level of the immediate
        ancestor of the individual on level  $i$  at time  $g$  */
    assert(i < POP_SIZE);
    assert(g < (tree.size() * POP_SIZE));
    assert(((g * POP_SIZE) + i) < tree.size());
    return (tree[(g * POP_SIZE) + i]);
}
```

This code is used in chunk 22.

4.11 check if sample tree is complete

check if a sample tree is complete, if we find a common ancestor of all the sampled leaves in the given population ancestry; when in generation g the goal is to carry out the steps

$$\begin{aligned} & (i, \dots, i, j, \dots, j, k, \dots, k) \\ & \rightarrow (a_i(g), \dots, a_i(g), a_j(g), \dots, a_j(g), a_k(g), \dots, a_k(g)) \\ & \rightarrow (a_i(g), a_j(g), a_k(g)) \\ & g \leftarrow g - 1 \end{aligned}$$

where $i = a_i(g + 1)$, $j = a_j(g + 1)$, $k = a_k(g + 1)$

15 $\langle \text{is sample tree complete? } 15 \rangle \equiv$

```
static bool allcoalesced ( const std::vector < std::pair < std::size_t , std::size_t >> &sample,
                          const std::vector < std::size_t > &tre ) {
    assert(POP_SIZE ≤ tre.size());
    assert((tre.size() % POP_SIZE) < 1);
    int g = static_cast<int>(tre.size()/POP_SIZE) - 1;
    assert(sample.size() ≡ SAMPLE_SIZE);
    std::vector < std::size_t > a(SAMPLE_SIZE, 0);    /*
        pair is (size of block) */                /*
        copy ancestor indexes into a; then work with a to check if tree is complete */
    std::transform ( sample.begin(), sample.end(), a.begin(), [](const auto &x)
    {
        return x.second;
    }
    ); while ((a.size() > 1) ∧ (g > -1)) {
        /* getagi § 4.10; (i, j, k) → (a_i(g), a_j(g), a_k(g)) */
        std::transform ( a.begin(), a.end(), a.begin(), [&g, &tre](const auto &i)
        {
            return getagi(g, i, tre);
        }
        ); /*
            remove duplicates from a; for example assuming i < j < k then
            (k, i, k, j, ..., i) → (i, j, k) */
        std::sort(a.begin(), a.end());
        a.erase(std::unique(a.begin(), a.end()), a.end());
        --g; } /*
            return True if the tree is complete */
    return (a.size() < 2); }
```

This code is used in chunk 22.

4.12 sort in descending order

sort the sample in descending order according to size of block; each element of sample is (size of block, level of block)

16 $\langle \text{bigger } 16 \rangle \equiv$

```
static bool comp ( const std::pair < std::size_t , std::size_t > &a, const std::pair <
    std::size_t , std::size_t > &b )
{
    return a.first > b.first;
}
```

This code is used in chunk 22.

4.13 update sample

here we update the sample over one generation and merge blocks with same immediate ancestor;
suppose $a'_1(g), \dots, a'_k(g)$ are the unique immediate ancestors

$$\begin{aligned} & \text{look up immediate ancestors } \{\dots, (s, i), \dots\} \rightarrow \{\dots, (s, a_i(g)), \dots\} \\ & \text{sum block sizes of blocks with same ancestor } \left\{ \dots, \left(\sum_j \mathbb{1}_{\{a_j(g)=a'_\ell(g)\}} s_j, a'_\ell(g) \right), \dots \right\} \end{aligned}$$

The goal here is to end up with the sample configuration

$$\left\{ \left(\sum_j \mathbb{1}_{\{a_j(g)=a'_1(g)\}} s_j, a'_1(g) \right), \dots, \left(\sum_j \mathbb{1}_{\{a_j(g)=a'_k(g)\}} s_j, a'_k(g) \right) \right\}$$

17 $\langle \text{merge blocks and update sample 17} \rangle \equiv$

```
static void updatesample ( const std::size_t &g, std::vector< std::pair< std::size_t ,
    std::size_t >> &sample, const std::vector< std::size_t > &tree )
{
    /*
        recall each sample 'block' is (size of block, level of block) */
    std::size_t s
    {}
    ;
    const std
        ::size_t e = sample.size();
    std::size_t k
    {}
    ;
    std::size_t t
    {}
    ;
    for (std::size_t i = 0; i < e; ++i) {
        s = 0;
        for (std::size_t j = i; j < e; ++j) {
            /*
                sum size of blocks with same immediate ancestor; getagi § 4.10 */
            s += (getagi(g, sample[i].second, tree) == getagi(g, sample[j].second,
                tree) ? sample[j].first : 0);
            /*
                if block has already been merged set size of block to zero */
            sample[j].first = (getagi(g, sample[i].second, tree) == getagi(g, sample[j].second,
                tree) ? 0 : sample[j].first);
        }
        if (s > 0) {
            /*
                record only a new merged block or a current block not merging */
            ++k;
            /*
                record the continuing block with the block size */
            /*
                and the level of the immediate ancestor of the block */
            sample.push_back(std::make_pair(s, getagi(g, sample[i].second, tree)));
        }
        t += s;
    }
    /*
        push blocks with size 0 to the back; comp § 4.12 */
    std::sort(sample.begin(), sample.end(), comp);
    /*
        k counts the new/continuing blocks */
    sample.resize(k);
}
```

```
    assert(sample.size() == k);  
    assert(t == SAMPLE_SIZE);  
    assert(sample.size() > 1 ? (sample.back().first < SAMPLE_SIZE) :  
           (sample.back().first == SAMPLE_SIZE));  
}
```

This code is used in chunk 22.

4.14 update $\ell_i^N(n)$

the goal here is to update the lengths $\ell_i^N(n)$; if s_1, \dots, s_m are the current block sizes then

$$\ell_s \leftarrow \ell_s + 1$$

for $s = s_1, \dots, s_m$

```
18  ⟨ update  $\ell_i^N(n)$  18 ⟩ ≡
    static void updatevbi ( const std::vector < std::pair < std::size_t ,
                          std::size_t >> &sample, std::vector < std::size_t > &vbi )
    {
        /*
           pair is (size of block, level of block) */
        for (std::size_t i = 0; i < sample.size(); ++i) {
            assert(sample[i].first > 0);
            assert(sample[i].first < SAMPLE_SIZE);
            ++vbi[0];
            ++vbi[sample[i].first];
        }
    }
```

This code is used in chunk 22.

4.15 update $r_i^N(n)$

given the lengths $\ell_1^N(n), \dots, \ell_{n-1}^N(n)$ of one complete tree update the approximation of $\mathbb{E} \left[\tilde{R}_i^N(n) \right]$

$$r_i^N(n) \leftarrow r_i^N(n) + \frac{\ell_i^N(n)}{\ell_1^N(n) + \dots + \ell_{n-1}^N(n)}$$

and it must hold that $\sum_j \ell_j^N(n) \geq n$

```
19  < update  $r_i^N(n)$  19 >  $\equiv$ 
    static void updatereestimate ( const std::vector < std::size_t > &b, std::vector <
        double > &r, const std::size_t SAMPLE_SIZE )
    {
        assert(b[0]  $\geq$  SAMPLE_SIZE);
        for (std::size_t i = 1; i < SAMPLE_SIZE; ++i) {
            r[i] += static_cast<double>(b[i])/static_cast<double>(b[0]);
        }
    }
```

This code is used in chunk 22.

4.16 one experiment

the goal here is to generate one complete sample tree and read the branch lengths off the tree and update the approximation of $\mathbb{E} [R_i^N(n)]$; recall Figure 1

20 \langle generate one experiment 20 $\rangle \equiv$

```
static void onexperiment ( std::vector < double > &vr, const std::vector <
    double > &vcmf_alpha, const std::vector < double > &vcmf_kappa ) {
    std::vector < std::pair < std::size_t , std::size_t >> sample
    {}
    ;
    std::vector < std::size_t > tre(POP_SIZE,0);
    std::iota(tre.begin(), tre.end(), 0);
    std::vector < std::size_t > currentbi(SAMPLE_SIZE,0);
    std::size_t g
    {}
    ; /*
        sample levels for the sample § 4.9 */
    randomsample(sample); /*
        § 4.11 */
    while (¬allcoalesced(sample, tre)) { /*
        add to the population ancestry §4.7 */
        MODEL ? addgeneration_modelone(tre,vcmf_alpha,
            vcmf_kappa) : addgeneration_modelnull(tre,
            (gsl_rng_uniform(rngtype) < EPSILON ? vcmf_alpha : vcmf_kappa));
        randomsample(sample);
        ++g;
    } /*
        we have a complete sample tree so read the branch lengths  $\ell_i^N(n)$  */
    while (sample.back().first < SAMPLE_SIZE) { /*
        update  $\ell_i^N(n)$  § 4.14 */
        updatevbi(sample, currentbi); /*
        merge blocks and update sample configuration § 4.13 */
        updatesample(g, sample, tre);
        --g;
    } /*
        given lengths  $\ell_i^N(n)$  update approximation of  $\mathbb{E} [\tilde{R}_i^N(n)]$  § 4.15 */
    updatereestimate(currentbi, vr, SAMPLE_SIZE); }
```

This code is used in chunk 22.

4.17 approximate $\mathbb{E} \left[\tilde{R}_i^N(n) \right]$

approximate $\mathbb{E} \left[\tilde{R}_i^N(n) \right]$ by sampling complete sample trees and reading branch lengths and averaging; recall Figure 2

21 $\langle \text{go ahead} - \text{approximate } \mathbb{E} \left[\tilde{R}_i^N(n) \right] \text{ 21} \rangle \equiv$

```

static void approximate_eri() {
    std::vector < double > vcmf_smalla(CUTOFF + 1); std::vector <
        double > vcmf_biga(CUTOFF + 1); std::vector < double > vR(SAMPLE_SIZE);    /*
        generate a cumulative mass function for sampling bounded numbers of potential
        offspring using (1); § 4.4 */
    generate_cmfs(vcmf_smalla, vcmf_biga);
    int r = EXPERIMENTS + 1;
    while (--r > 0) { /*
        § 4.16 */
        onexperiment(vR, vcmf_smalla, vcmf_biga);
    } /*
        return  $r_i^N(n)$  summed over the experiments */
    for (const auto &z:vR)
    {
        std::cout << z << '\n';
    }
}

```

This code is used in chunk 22.

4.18 the main module

```

22      /*
      § 4.1 */
      <included libraries 5>      /*
      § 4.2 */
      <random number generators 6>      /*
      § 4.3 */
      <mass function 7>      /*
      § 4.4 */
      <cmf for number of offspring 8>      /*
      § 4.5 */
      <one bounded number 9>      /*
      § 4.6 */
      <an unbounded  $X$  10>      /*
      § 4.7 */
      <one generation 11>      /*
      § 4.8 */
      <add a generation for model 1 12>      /*
      § 4.9 */
      <random sample 13>      /*
      § 4.10 */
      <get immediate ancestor 14>      /*
      § 4.11 */
      <is sample tree complete? 15>      /*
      § 4.12 */
      <bigger 16>      /*
      § 4.13 */
      <merge blocks and update sample 17>      /*
      § 4.14 */
      <update  $\ell_i^N(n)$  18>      /*
      § 4.15 */
      <update  $r_i^N(n)$  19>      /*
      § 4.16 */
      <generate one experiment 20>      /*
      § 4.17 */
      <go ahead – approximate  $\mathbb{E}[\tilde{R}_i^N(n)]$  21>
int main(int argc, const char *argv[])
{
    /*
    input random seed § 4.2 */
    setup_rng(static_cast<std::size_t>(atoi(argv[1])));      /*
    § 4.17 */
    approximate_eri();
    gsl_rng_free(rngtype);
    return 0;
}

```

5 conclusion and bibliography

Gene genealogies are generated as the population evolves forward in time (Figure 4). We evolve a haploid panmictic population of constant size forward in time and record the ancestry (Figure 3). We read the branch lengths off independent fixed complete sample trees to approximate $\mathbb{E}[\tilde{R}_i^N(n)]$. In Figure 5 are examples of approximations of $\mathbb{E}[\tilde{R}_i^N(n)]$.

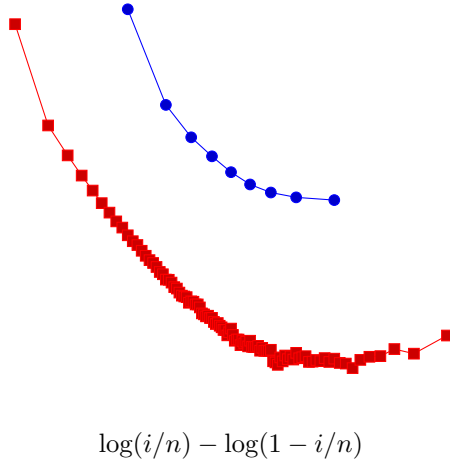


Figure 5: *An example approximation of $\mathbb{E}[\tilde{R}_i(n)]$ for the given parameter values and graphed as logits against $\log(i/n) - \log(1 - i/n)$ for $i = 1, 2, \dots, n - 1$ where n is sample size*

Here we consider the simplest case of a single haploid panmictic population of constant size. The population may evolve according to sweepstakes reproduction (heavy-tailed offspring number distribution). We model sweepstakes reproduction through the law (1) on the number of potential offspring produced independently by the current individuals; (1) is an extension of the one in [Sch03]. There is evidence that predictions of genetic diversity in populations evolving according to sweepstakes reproduction may depend on how one views gene genealogies [D2024]

References

- [D2024] Diamantidis, Dimitrios and Fan, Wai-Tong (Louis) and Birkner, Matthias and Wakeley, John. Bursts of coalescence within population pedigrees whenever big families occur. *Genetics* Volume 227, February 2024.
<https://dx.doi.org/10.1093/genetics/iyae030>.
- [CDEH25] JA Chetwyn-Diggle, Bjarki Eldon, and Matthias Hammer. Beta-coalescents when sample size is large. In preparation, 2025+.
- [DK99] P Donnelly and T G Kurtz. Particle representations for measure-valued population models. *Ann Probab*, 27:166–205, 1999.
<https://dx.doi.org/10.1214/aop/1022677258>
- [KL94] Donald Ervin Knuth and Silvio Levy. *The CWEB system of structured documentation: version 3.0*. Addison-Wesley Longman Publishing Co., Inc., Reading, Massachusetts, 1994.
- [KR88] Brian W Kernighan and Dennis M Ritchie. *The C programming language*, 1988.
- [Pit99] J Pitman. Coalescents with multiple collisions. *Ann Probab*, 27:1870–1902, 1999.
<https://dx.doi.org/10.1214/aop/1022874819>
- [Sag99] S Sagitov. The general coalescent with asynchronous mergers of ancestral lines. *J Appl Probab*, 36:1116–1125, 1999.
<https://doi.org/10.1239/jap/1032374759>

- [Sch03] J Schweinsberg. Coalescent processes obtained from supercritical Galton-Watson processes. *Stoch Proc Appl*, 106:107–139, 2003.
[https://doi.org/10.1016/S0304-4149\(03\)00028-0](https://doi.org/10.1016/S0304-4149(03)00028-0)
- [Tan11] O Tange. GNU parallel – the command-line power tool. The USENIX Magazine, 2011.

Index

a: [7](#).
abs: [8](#).
addgeneration_modelnull: [11](#), [20](#).
addgeneration_modelone: [12](#), [20](#).
allcoalesced: [15](#), [20](#).
ALPHA: [7](#), [8](#), [10](#).
approximate_eri: [21](#), [22](#).
argc: [22](#).
argv: [22](#).
assert: [8](#), [9](#), [11](#), [12](#), [13](#), [14](#), [15](#), [17](#), [18](#), [19](#).
atoi: [22](#).
back: [8](#), [17](#), [20](#).
begin: [8](#), [11](#), [12](#), [13](#), [15](#), [17](#), [20](#).
clear: [13](#).
cmfa: [8](#).
cmfA: [8](#).
comp: [16](#), [17](#).
cout: [21](#).
currentbi: [20](#).
CUTOFF: [8](#), [9](#), [21](#).
d: [8](#).
e: [11](#), [12](#), [17](#).
end: [8](#), [11](#), [12](#), [13](#), [15](#), [17](#), [20](#).
EPSILON: [12](#), [20](#).
erase: [15](#).
EXPERIMENTS: [21](#).
first: [16](#), [17](#), [18](#), [20](#).
floor: [10](#).
g: [14](#), [15](#), [17](#), [20](#).
generate_cmfs: [8](#), [21](#).
getagi: [14](#), [15](#), [17](#).
gN: [14](#).
gsl_rng: [6](#).
gsl_rng_alloc: [6](#).
gsl_rng_default: [6](#).
gsl_rng_env_setup: [6](#).
gsl_rng_free: [22](#).
gsl_rng_set: [6](#).
gsl_rng_type: [6](#).
gsl_rng_uniform: [20](#).
gsl_rng_uniform_pos: [9](#), [10](#), [12](#).
i: [8](#), [11](#), [12](#), [13](#), [14](#), [15](#), [17](#), [18](#), [19](#).
insert: [11](#), [12](#).
iota: [13](#), [20](#).
j: [9](#), [17](#).
k: [7](#), [17](#).
KAPPA: [7](#), [8](#).
kernel: [7](#), [8](#).
main: [22](#).
make_pair: [13](#), [17](#).
MODEL: [1](#), [20](#).
mt19937_64: [6](#).
onexperiment: [20](#), [21](#).
pair: [13](#), [15](#), [16](#), [17](#), [18](#), [20](#).
POP_SIZE: [11](#), [12](#), [13](#), [14](#), [15](#), [20](#).
pow: [7](#), [10](#).
push_back: [13](#), [17](#).
r: [21](#).
random_device: [6](#).
randomsample: [13](#), [20](#).
randomseed: [6](#).
randomX: [9](#), [11](#), [12](#).
reserve: [11](#), [12](#), [13](#).
resize: [11](#), [12](#), [17](#).
rng: [6](#), [11](#), [12](#), [13](#).
rngtype: [6](#), [9](#), [10](#), [12](#), [20](#), [22](#).
s: [6](#), [8](#), [17](#).
sample: [13](#), [15](#), [17](#), [18](#), [20](#).
SAMPLE_SIZE: [13](#), [15](#), [17](#), [18](#), [19](#), [20](#), [21](#).
second: [15](#), [17](#).
setup_rng: [6](#), [22](#).
shuffle: [11](#), [12](#), [13](#).
size: [11](#), [12](#), [13](#), [14](#), [15](#), [17](#), [18](#).
sort: [15](#), [17](#).
std: [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#), [19](#), [20](#), [21](#), [22](#).
T: [6](#).
t: [17](#).
transform: [8](#), [15](#).
tre: [15](#), [20](#).
tree: [11](#), [12](#), [14](#), [17](#).
u: [9](#), [12](#).
unboundedrandomX: [10](#).
unique: [15](#).
updaterestimate: [19](#), [20](#).
updatesample: [17](#), [20](#).
updatevbi: [18](#), [20](#).
vbi: [18](#).
vcmf: [11](#).
vcmf_a: [12](#).
vcmf_A: [12](#).
vcmf_alpha: [20](#).
vcmf_biga: [21](#).
vcmf_kappa: [20](#).
vcmf_smalla: [21](#).
vector: [8](#), [9](#), [11](#), [12](#), [13](#), [14](#), [15](#), [17](#), [18](#), [19](#), [20](#), [21](#).
vr: [20](#).
vR: [21](#).
x: [8](#), [11](#), [12](#), [15](#).
z: [21](#).

List of Refinements

- $\langle \text{add a generation for model 1 } 12 \rangle$ Used in chunk 22.
- $\langle \text{an unbounded } X \text{ } 10 \rangle$ Used in chunk 22.
- $\langle \text{bigger } 16 \rangle$ Used in chunk 22.
- $\langle \text{cmf for number of offspring } 8 \rangle$ Used in chunk 22.
- $\langle \text{generate one experiment } 20 \rangle$ Used in chunk 22.
- $\langle \text{get immediate ancestor } 14 \rangle$ Used in chunk 22.
- $\langle \text{go ahead – approximate } \mathbb{E} \left[\tilde{R}_i^N(n) \right] \text{ } 21 \rangle$ Used in chunk 22.
- $\langle \text{included libraries } 5 \rangle$ Used in chunk 22.
- $\langle \text{is sample tree complete? } 15 \rangle$ Used in chunk 22.
- $\langle \text{mass function } 7 \rangle$ Used in chunk 22.
- $\langle \text{merge blocks and update sample } 17 \rangle$ Used in chunk 22.
- $\langle \text{one bounded number } 9 \rangle$ Used in chunk 22.
- $\langle \text{one generation } 11 \rangle$ Used in chunk 22.
- $\langle \text{random number generators } 6 \rangle$ Used in chunk 22.
- $\langle \text{random sample } 13 \rangle$ Used in chunk 22.
- $\langle \text{update } \ell_i^N(n) \text{ } 18 \rangle$ Used in chunk 22.
- $\langle \text{update } r_i^N(n) \text{ } 19 \rangle$ Used in chunk 22.