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

Bjarki Eldon[1] [2]

With this C++ code one estimates mean conditional relative branch lengths $\rho_i^N(n) \equiv \mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ where $R_i^N(n) = L_i^N(n)/\sum_{j=1}^{2n-1} L_j^N(n)$ and $L_i^N(n)$ is the random length of branches supporting $i \in \{1, 2, \ldots, 2n-1\}$ leaves where n is the number of diploid individuals sampled (hence $2n$ gene copies); $\mathcal{A}^{(N,n)}$ is the sigma field keeping track of the ancestral relations of whole population. The idea is that the gene copies of a sample share an ancestry, a gene genealogy, even if the genealogy is unknown. The quantity $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ is mean relative branch length conditional on the population ancestry, the ancestral relations of all gene copies at all times. The sample comes from a finite diploid panmictic population of constant size evolving according to specific models of sweepstakes reproduction

Contents

1 Copyright 2

2 Compilation, output and execution 3

3 intro 4

4 code 6
- 4.1 includes . . . 7
- 4.2 constants . . . 8
- 4.3 the random number generators . . . 9
- 4.4 the probability mass function for number of offspring . . . 10
- 4.5 generate a cumulative mass function . . . 11
- 4.6 sample a random number of potential offspring . . . 12
- 4.7 assign chromosomes to offspring . . . 13
- 4.8 add generation to population ancestry . . . 14
- 4.9 a random sample . . . 15
- 4.10 get level of immediate ancestor . . . 16
- 4.11 check tree completeness . . . 17
- 4.12 get a complete sample tree . . . 18
- 4.13 update the site-frequency spectrum . . . 19
- 4.14 update estimates $\widehat{r}_i^N(n)$ of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ . . . 20
- 4.15 read a complete tree . . . 21
- 4.16 approximate $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ . . . 22
- 4.17 the main module . . . 23

5 conclusions and bibliography 24

[1] beldon11@gmail.com

[2] Supported by Deutsche Forschungsgemeinschaft (DFG) - Projektnummer 273887127 through DFG SPP 1819: Rapid Evolutionary Adaptation grant STE 325/17 to Wolfgang Stephan; acknowledge funding by the Icelandic Centre of Research (Rannís) through an Icelandic Research Fund (Rannsóknasjóður) Grant of Excellence no. 185151-051 to Einar Árnason, Katrín Halldórsdóttir, Alison M. Etheridge, Wolfgang Stephan, and BE. BE also acknowledges Start-up module grants through SPP 1819 with Jere Koskela and Maite Wilke-Berenguer, and with Iulia Dahmer.
January 9, 2026

1 Copyright

Copyright © 2026 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/.

2 Compilation, output and execution

This CWEB [3] document (the `.w` file) can be compiled with `cweave` to generate a `.tex` file, and with `ctangle` to generate a `.c` [2] file.

One can use `cweave` to generate a `.tex` file, and `ctangle` to generate a `.c` file. To compile the C++ code (the `.c` file), one needs the GNU Scientific Library.

Compiles on `Linux debian` 6.12.9-amd64 with `ctangle` 4.11 and `g++` 14.2 and `GSL` 2.8

Using a Makefile can be helpful, naming this file `iguana.w`

```
iguana.pdf :  iguana.tex
     cweave iguana.w
     pdflatex iguana
     bibtex iguana
     pdflatex iguana
     pdflatex iguana
     ctangle iguana
     g++ -std=c++26 -O3 -march=native -m64 -xc++ iguana.c -lm -lgsl -lgslcblas

clean :
     rm -vf iguana.c iguana.tex
```

Use `valgrind` to check for memory leaks:

```
valgrind -v --leak-check=full --leak-resolution=high --num-callers=40 --vgdb=full
<program call>
```

Use `cppcheck` to check the code:

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

To generate estimates on a computer with several CPUs it may be convenient to put in a text file (`simfile`):

```
./a.out $(shuf -i <smallest random seed>-<largest random seed> -n1) > resout<i>
```

for $i = 1, \ldots, y$ for suitables choices of random seeds so that no two replications will have the same seed and use `parallel`[5]

```
parallel --gnu -jy ::::  ./simfile
```

3 intro

Suppose a population is and evolves as in Definition 3.1.

Definition 3.1 (Evolution of a diploid population) *Consider a diploid (each diploid individual carries two gene copies or chromosomes; each diploid individual can also be seen as a pair of chromosomes) panmictic population of constant size $2N$ diploid individuals. In any given generation we arbitrarily label each individual with a unique label, and form all possible $(2N-1)N$ (unordered) pairs of labels. Then we sample N pairs of labels independently and uniformly at random without replacement. The N parent pairs thus formed independently produce random numbers of diploid potential offspring according to some given law. Each offspring receives two chromosomes, one chromosome from each of its two parents, with each inherited chromosome sampled independently and uniformly at random from among the two parent chromosomes. If the total number of potential offspring is at least $2N$, we sample $2N$ of them uniformly at random without replacement to survive to maturity and replace the current individuals; otherwise we assume the population is unchanged over the generation (all the potential offspring perish before reaching maturity).*

Remark 3.2 (Illustrating Definition 3.1) *The mechanism described in Definition 3.1 is illustrated below, where $\{a,b\}$ denotes a diploid individual carrying gene copies a, b and $\{\{a,b\},\{c,d\}\}$ denotes a pair of parents. Here we have arbitrarily labelled the gene copies just for the sake of illustrating the evolution over one generation.*

stage	*individuals involved*
1	$\{\{a,b\},\{c,d\}\},\ldots,\{\{w,x\},\{y,z\}\}$: *N parent pairs*
2	$\underbrace{\{a,d\},\{b,d\},\ldots,\{a,c\}}_{X_1},\ldots,\underbrace{\{w,y\},\ldots,\{x,z\}}_{X_N}$: *$X_1+\cdots+X_N$ potential offspring*
3	$\{b,d\},\ldots,\{x,y\}$: *$2N$ surviving offspring (whenever $X_1+\cdots+X_N \geq 2N$)*

In stage 1 above, the current $2N$ diploid individuals randomly form N pairs; in stage 2 the N pairs formed in stage 1 independently produce random numbers $X_1,\ldots,X_N$ of potential offspring, where each offspring receives one gene copy (or chromosome) from each of its two parents; in the third stage $2N$ of the $X_1+\cdots+X_N$ potential offspring (conditional on there being at least $2N$ of them) are sampled uniformly and without replacement to survive to maturity and replace the parents.

Write $[n] \equiv \{1,2,\ldots,n\}$ for $n \in \mathbb{N} \equiv \{1,2,\ldots\}$. Let $\left\{\xi^{n,N}(r) : r \in \mathbb{N}\cup\{0\}\right\}$ denote the *ancestral process*, a Markov sequence tracking the ancestral relations of sampled gene copies in a finite population. Let $\{\xi^n(t) : t \geq 0\}$ denote a coalescent, a Markov chain on the partitions of $[2n]$. Let $\#A$ be the cardinality of a given set A, write $\tau^N(n) := \inf\left\{j \in \mathbb{N} : \#\xi^{n,N}(j) = 1\right\}$, and $\tau(n) := \inf\{t \geq 0 : \#\xi^n(t) = 1\}$. Consider the functionals, for $i \in [2n-1]$,

$$L_i^N(n) := \sum_{j=1}^{\tau^N(n)} \#\left\{\xi \in \xi^{n,N}(j) : \#\xi = i\right\}, \quad L^N(n) := \sum_{j=1}^{\tau^N(n)} \#\xi^{n,N}(j)$$

$$L_i(n) := \int_0^{\tau(n)} \#\left\{\xi \in \xi^n(t) : \#\xi = i\right\} dt, \quad L(n) := \int_0^{\tau(n)} \#\xi^n(t)dt$$

[1]. The functionals $L_i^N(n)$ and $L_i(n)$ are the random length of branches supporting $i \in [2n-1]$ leaves, $L^N(n) = L_1^N(n) + \cdots + L_{n-1}^N(n)$, and $L(n) = L_1(n) + \cdots + L_{n-1}(n)$.

Write $R_i^N(n) \equiv L_i^N(n)/\sum_{j=1}^{2n-1} L_j^N(n)$. We estimate $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ when the sample comes from a finite haploid panmictic population evolving according to sweepstakes reproduction (heavy-tailed offspring number distribution).

Let $X_1, \ldots, X_N$ denote the random number of potential offspring produced by the current individuals. They are independent but may not always be identically distributed. Let X be independent of $X_1, \ldots, X_N$ and suppose

$$\mathbb{P}\left(X = k\right) = C\left(\frac{1}{k^a} - \frac{1}{(1+k)^a}\right), \quad k \in \{2, 3, \ldots, \zeta(N)\} \tag{1}$$

with $a > 0$ and where $\mathbb{P}\left(X \in \{2, 3, \ldots, \zeta(N)\}\right) = 1$. Write $X \rhd L(a, \zeta(N))$ when X is distributed according to (1) for some given a and $\zeta(N)$. In any given generation the current individuals independently produce potential offspring according to (1); from the pool of potential offspring N of them are sampled uniformly at random and without replacement to survive and reach maturity and replace the current individuals. Note that (1) is an extension of [4, Equation 11].

Let $0 < \alpha < 2$ and $\kappa \geq 2$ be fixed. We consider two models; one where $X_1, \ldots, X_N$ are iid and in generation g with $(U_g)_g$ a sequence of iid uniforms

$$X_1 \rhd L\left(\mathbb{1}_{\{U_g \leq \varepsilon_N\}}\alpha + \mathbb{1}_{\{U_g > \varepsilon_N\}}\kappa, \zeta(N)\right) \tag{2}$$

We also consider another model where the $X_1, \ldots, X_N$ are independent, X_i is as in (2) for i picked uniformly at random and $X_{j \neq i} \rhd L(\kappa, \zeta(N))$. In this second model the $X_1, \ldots, X_N$ are independent but may not always be identically distributed. For suitable choices of ε_N one can identify the limiting diffusions/coalescents. Clearly, choosing $\varepsilon_N \geq 1$ results in iid $X_1, \ldots X_N$ with $X_1 \rhd L(\alpha, \zeta(N))$; taking $\varepsilon_N \leq 0$ gives an alternative to the Wright-Fisher model.

The quantity $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ is the mean relative branch length conditioned on the population ancestry (the ancestral relations of all the gene copies in the population at all times). The idea is that the gene copies in a sample are related through a single tree, a gene genealogy, even if the tree is unknown. We approximate $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ by averaging over population ancestries.

The algorithm is summarized in § 4, the code follows in § 4.1–§ 4.17, we conclude in § 5. Comments within the code are in **this font and color**

4 code

The ancestry $\mathbb{A} = (A_i(g))_{i,g}$ records the ancestral relations of the gene copies (chromosomes); if the chromosome on level ℓ at time g produces k surviving copies then $A_{j_1}(g+1) = \ldots = A_{j_k}(g+1) = \ell$; and if $A_\ell(g) = A_k(g)$ then the individuals living on level ℓ resp. k at time g share an immediate ancestor. The ancestry $\mathbb{A}$ records the ancestral relations of $4N$ gene copies for each generation, where N is the number of parent pairs producing potential offspring.

1. $\left(\widehat{r}_i^N(n), \ldots, \widehat{r}_{2n-1}^N(n)\right) \leftarrow (0, \ldots, 0)$
2. For each of M experiments § 4.16 :
   (a) initialise the ancestry $A_\ell(0) = \ell$ for $\ell = 1, 2, \ldots, 4N$
   (b) $\left(\ell_1^N(n), \ldots, \ell_{n-1}^N(n)\right) \leftarrow (0, \ldots, 0)$
   (c) **while** a sample tree is incomplete § 4.12 :
      i. add to the ancestry $\mathbb{A}$ § 4.8 by sampling a pool of potential offspring using § 4.6 and assigning gene copies (chromosomes) to each offspring § 4.7
      ii. take a random sample § 4.9
      iii. check if the sample is complete § 4.11
   (d) given a complete sample tree read the branch lengths $\ell_i^N(n)$ off the fixed tree § 4.15
   (e) update estimate $\widehat{r}_i^N(n)$ of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ § 4.14

$$\widehat{r}_i^N(n) \leftarrow \frac{\ell_i^N(n)}{\sum_j \ell_j^N(n)} + \widehat{r}_i^N(n)$$

3. return the estimates $(1/M)\widehat{r}_i^N(n)$ of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$

4.1 includes

the included libraries; we use the GSL library

5 ⟨ includes 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 <unordered_set>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_math.h>
```

This code is used in chunk 21.

4.2 constants

the parameter values

6 ⟨ constants 6 ⟩ ≡

```
/*
    N diploid individuals so 2N gene copies */
const std
    ::size_t CONST_N = 1 · 10^2; const std
        ::size_t CONST_POP_SIZE = 2 * CONST_N;
        const double dN = static_cast⟨double⟩(CONST_N);    /*
            the upper bound ζ(N) */
        const std
            ::size_t CONST_CUTOFF = CONST_POP_SIZE *
                static_cast⟨std::size_t⟩(ceil(log(dN)));    /*
                α */
            const double CONST_ALPHA = 1.01;    /*
                κ */
            const double CONST_A = 2.0;    /*
                ε_N the probability of favorable conditions */
            const double CONST_VAREPSILON = 0.1;    /*
                sample size n of diploid individuals then 2n gene copies in sample */
            const std
                ::size_t CONST_SAMPLE_SIZE = 1 · 10^1;
                const int CONST_EXPERIMENTS = 1;
```

This code is used in chunk 21.

4.3 the random number generators

the STL and GSL random number generators

7 ⟨ rngs 7 ⟩ ≡

```
/*
    obtain a seed out of thin air for the random number engine */
std::random_device randomseed;    /*
    Standard Mersenne twister random number engine seeded with randomseed() */
std::mt19937_64 rng(randomseed());
gsl_rng * rngtype;
static 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 21.

4.4 the probability mass function for number of offspring

the probability mass function (1)

8 ⟨ pmf 8 ⟩ ≡

```
static double pmf ( const std ::size_t &k, const double &a )
        {     /*
                return k^{-a} - (1 + k)^{-a} */
          return (pow(1./static_cast⟨double⟩(k),
                a) - pow(1./static_cast⟨double⟩(k + 1), a));
        }
```

This code is used in chunk 21.

4.5 generate a cumulative mass function

generate a cumulative mass function for sampling numbers of potential offspring

9 ⟨ generate cmf 9 ⟩ ≡

```
static void generatecmf ( std::vector < double > &cmfalpha, std::vector <
    double > &cmfA ) {    /*
    the containers are initialised to (0, ..., 0)  */
double salpha
{ }
;
double sA
{ }
;
for (std::size_t i = 2; i ≤ CONST_CUTOFF; ++i) {    /*
      pmf § 4.4  */
  cmfalpha[i] = cmfalpha[i − 1] + pmf (i, CONST_ALPHA);
  cmfA[i] = cmfA[i − 1] + pmf (i, CONST_A);
  salpha += pmf (i, CONST_ALPHA);
  sA += pmf (i, CONST_A);
}
assert(salpha > 0.);
assert(sA > 0.); std::transform (cmfalpha.begin( ), cmfalpha.end( ), cmfalpha.begin( ),
        [&salpha](const auto &x)
{
  return x/salpha;
}
) ; std::transform (cmfA.begin( ), cmfA.end( ), cmfA.begin( ), [&sA](const auto &x)
{
  return x/sA;
}
) ;
cmfalpha[CONST_CUTOFF] = 1.;
cmfA[CONST_CUTOFF] = 1.; }
```

This code is used in chunk 21.

4.6 sample a random number of potential offspring

sample one random number of potential offspring $\min\{j : u \leq F(j)\}$ where u a random uniform and F the cumulative mass function

10 ⟨ get one random number of potential offspring 10 ⟩ ≡

```
static std ::size_t randomX ( const std::vector < double > &f )
    {
      const double u = gsl_rng_uniform_pos(rngtype);
      std::size_t j
      {2};
      while (u > f[j]) {
        ++j;
      }
      assert(j ≥ 2);
      return j;
    }
```

This code is used in chunk 21.

4.7 assign chromosomes to offspring

assign chromosomes to potential offspring; there are N parent pairs so $2N$ diploid individuals and $4N$ gene copies; then parent pair i will have chromosomes labelled $4i, 4i+1, 4i+2, 4i+3$ to distribute among its offspring; the ancestry will record the ancestral relations of the gene copies, or chromosomes

11 ⟨ assign chromosomes to offspring 11 ⟩ ≡

```
static void assignchromstooffspring ( const std ::size_t &i, const std ::size_t
        &x, std::vector < std::pair < std::size_t , std::size_t >> &y )
{    /*
       i is the parent pair index from 0 to N − 1; each parent pair has 4
       chromosomes; need y as vector of pairs since will eventually remove
       some of them; each potential offspring is pair of gene copies; since will
       eventually remove some of them; 0 ≤ 4i ≤ 4N − 4 */
  for (std::size_t k = 0; k < x; ++k) {
    y.push_back(std::make_pair((4 ∗ i) + (gsl_rng_uniform_pos(rngtype) <
        0.5 ? 0 : 1), (4 ∗ i) + (gsl_rng_uniform_pos(rngtype) < 0.5 ? 2 : 3)));
  }
}
```

This code is used in chunk 21.

4.8 add generation to population ancestry

add a new set of surviving offspring to population ancestry; the ancestry records the ancestral relations of gene copies

12 ⟨ add a new generation 12 ⟩ ≡

```
static void addgeneration (std::vector < std::size_t > &tree, const std::vector <
    double > &vcmf ) {      /*
    N parent pairs produce offspring; 2N diploid individuals in the population */
std::size_t x
{ }
;
std::size_t s
{ }
; std::vector < std::pair < std::size_t , std::size_t >> y
{ }
;
y.clear( );      /*
    CONST_N parent pairs produce potential offspring § 4.2 */
for (std::size_t i = 0; i < CONST_N; ++i) {      /*
        x is number of diploid potential offspring produced by parent pair i; § 4.6 */
  x = randomX(vcmf);
  s += x;
  y.reserve(y.size( ) + x);      /*
        § 4.7 */
  assignchromstooffspring(i, x, y);
}      /*
        y is the record of diploid potential offspring */
assert(y.size( ) ≥ CONST_POP_SIZE);      /*
    we sample surviving offspring by shuffling an index vector V and recording the offspring
    corresponding to the first N elements of the shuffled vector */
std::vector < std::size_t > V(s, 0);
std::iota(V.begin( ), V.end( ), 0);      /*
    rng § 4.3 */
std::shuffle(V.begin( ), V.end( ), rng);      /*
    we want to add the ancestral relations of 4N gene copies to the ancestry */
tree.reserve(tree.size( ) + (4 * CONST_N));      /*
    add new generation to tree */      /*
    CONST_POP_SIZE is 2N the number of diploid individuals */
for (std::size_t z = 0; z < CONST_POP_SIZE; ++z) {
  tree.push_back(y[V[z]].first);
  tree.push_back(y[V[z]].second);
}
}
```

This code is used in chunk 21.

4.9 a random sample

get a random sample of diploid individuals by sampling a random set of levels

13 ⟨ a random sample 13 ⟩ ≡

```
static void randomsample(std::vector < std::size_t > &V,
        std::vector < std::size_t > &sample)
{    /*
      pair is (size of block, level of immediate ancestor of block) */    /*
      sample levels ranges from 0 to 4N − 2 */    /*
      the elements of V are in {0, 2, 4, . . . , 4N − 2} */
  std::shuffle(V.begin( ), V.end( ), rng);
  assert(sample.size( ) ≡ CONST_SAMPLE_SIZE);
  std::copy(V.begin( ), V.begin( ) + CONST_SAMPLE_SIZE, sample.begin( ));
}
```

This code is used in chunk 21.

4.10 get level of immediate ancestor

look up the immediate ancestor of the individual living on level ℓ at time g

14 ⟨ immediate ancestor 14 ⟩ ≡

```
static std ::size_t getagi ( const std ::size_t &g, const std ::size_t &l, const std::vector
                             < std::size_t > &t )
{     /*
        0 ≤ i < 4N */      /*
        each tree 'row' or generation is with indexes 0, 1, 2, . . . , 4N − 1
        */      /*
        CONST_POP_SIZE is 2N the number of diploid individuals */
  assert(((2 ∗ g ∗ CONST_POP_SIZE) + l) < t.size( ));
  return t[(2 ∗ g ∗ CONST_POP_SIZE) + l];
}
```

This code is used in chunk 21.

4.11 check tree completeness

checking if sample tree is incomplete; the tree is complete if the leaves have a common ancestor

15 ⟨ incomplete 15 ⟩ ≡

```
static bool treeincomplete (std::vector < std::size_t > &sample, const std::vector <
        std::size_t > &ancestry ) { std::unordered_set < std::size_t > s
{ }
;
s.clear( );
assert(s.empty( ));      /*
    sample has elements from {0, 2, 4, . . . , 4N − 2} since there are 2N diploid individuals; the
    indexes of diploid individuals; diploid individual on level ℓ has chromosomes on levels ℓ
    and ℓ + 1 */
std::for_each (sample.begin( ), sample.end( ), [&s](const auto &y)
{
  s.insert(y);
  s.insert(y + 1);
}
) ;
std::size_t g = (ancestry.size( )/(2 * CONST_POP_SIZE)) − 1;
std::vector < std::size_t > a(2 * CONST_SAMPLE_SIZE);
assert(a.size( ) ≡ s.size( )); while ((s.size( ) > 1) ∧ (g > 0)) { a.resize(s.size( )); std::transform
        (s.begin( ), s.end( ), a.begin( ), [&g, &ancestry](const auto &i)
{
  return getagi(g, i, ancestry);
}
) ;
s.clear( );
assert(s.empty( ));      /*
    s being an unordered set only records unique elements */
std::for_each (a.begin( ), a.end( ), [&s](const auto &b)
{
  s.insert(b);
}
) ;
assert(s.size( ) > 0);
−−g; } return s.size( ) > 1; }
```

This code is used in chunk 21.

4.12 get a complete sample tree

get a complete sample tree by adding to the ancestry and drawing new samples until a complete sample tree is found

16 ⟨ one complete sample tree 16 ⟩ ≡

```
static void getcompletesampletree (std::vector < std::size_t > &ancestry,
    std::vector < std::size_t > &sampledlevels, std::vector < std::size_t > &__V, const
    std::vector < double > &vcmf )
{   /*
      get a complete sample tree
      sample levels until a complete tree is found
      tree is population tree */   /*
      start a new population from scratch */
  ancestry.clear( );
  ancestry.reserve(2 * CONST_POP_SIZE);
  std::vector < std::size_t > (2 * CONST_POP_SIZE).swap(ancestry);   /*
      initialise A_ℓ(0) = ℓ for ℓ = 0, 1, ..., N − 1 */
  std::iota(ancestry.begin( ), ancestry.end( ), 0);   /*
      § 4.8 */
  addgeneration(ancestry, vcmf);   /*
      § 4.9 */
  randomsample(__V, sampledlevels);
  std::size_t number_generation = 1;   /*
      § 4.11 */
  while (treeincomplete(number_generation, sampledlevels, ancestry)) {
    addgeneration(ancestry, vcmf);
    randomsample(__V, sampledlevels);
    ++number_generation;
  }
}
```

This code is used in chunk 21.

4.13 update the site-frequency spectrum

update the site-frequency spectrum (branch lengths) $\ell_i^N(n)$ given current block sizes

17 ⟨ update the site-frequency spectrum 17 ⟩ ≡

```
static void updatesfs ( std::vector < double > &__ells, std::unordered_map < std::size_t
    , std::size_t > &__m ) {
std::for_each (__m.begin( ), __m.end( ), [&__ells](const auto &x)
{
  assert(x.second > 0);
  assert(x.second < 2 * CONST_SAMPLE_SIZE);
  ++__ells[0];
  ++__ells[x.second];
}
) ; }
```

This code is used in chunk 21.

4.14 update estimates $\widehat{r}_i^N(n)$ of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$

given branch lengths $\ell_i^N(n)$ update estimates $\widehat{r}_i^N(n)$ of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$

$$\widehat{r}_i^N(n) \leftarrow \widehat{r}_i^N(n) + \frac{\ell_i^N(n)}{\sum_{j=1}^{2n-1} \ell_j^N(n)}$$

18 ⟨ update $\widehat{r}_i^N(n)$ 18 ⟩ ≡

```
static void updaterrs ( std::vector < double > &__errs, const std::vector <
        double > &__ells ) {
  assert(__ells[0] > 0);
  const double d = __ells[0];
  std::transform (__ells.begin(), __ells.end(), __errs.begin(), __errs.begin(),
        [&d](const auto &x, const auto &y)
  {
    return y + (x/d);
  }
  ) ; }
```

This code is used in chunk 21.

4.15 read a complete tree

given the leaves of a complete tree read the branch lengths off the tree

19 ⟨ read a complete tree 19 ⟩ ≡

```
static void readcompletetree (std::vector < std::size_t > &sl, std::vector < double > &errs,
        const std::vector < std::size_t > &__ancestry ) {
    std::unordered_map < std::size_t , std::size_t > m
    { }
    ;
    m.clear( );
    std::unordered_map < std::size_t , std::size_t > tmp
    { }
    ;
    tmp.clear( );
    std::vector < double > ells(2 * CONST_SAMPLE_SIZE);
    std::size_t __g = (__ancestry.size( )/(2 * CONST_POP_SIZE)) - 1;      /*
        initialise the map m with 2n blocks of size 1 each labelled with the sampled levels  */
    std::for_each (sl.begin( ), sl.end( ), [&m](const auto &x)
    {
      m[x] = 1;
      m[x + 1] = 1;
    }
    ) ;
    assert(m.size( ) ≡ 2 * sl.size( )); while (m.size( ) > 1) {      /*
        updatesfs § 4.13  */
    updatesfs(ells, m);
    tmp.clear( );
    assert(tmp.size( ) < 1);      /*
        read new (merged and continuing) blocks into tmp; use getagi § 4.10  */
    std::for_each (m.begin( ), m.end( ), [&tmp, &__g, &__ancestry](const auto &x)
    {
      tmp[getagi(__g, x.first, __ancestry)] += x.second;
    }
    ) ;
    m.clear( );
    assert(m.size( ) < 1);
    m = tmp;
    assert(m.size( ) ≡ tmp.size( ));
    --__g; }      /*
        updaterrs § 4.14  */
    updaterrs(errs, ells); }
```

This code is used in chunk 21.

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

estimate $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ for a given number of CONST_EXPERIMENTS § 4.2

20 ⟨ go ahead — get $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ 20 ⟩ ≡

```
static void estimate( ) {
    std::vector < std::size_t > ancestry__
    { }
    ;
    std::vector < std::size_t > sl__(CONST_SAMPLE_SIZE);
    std::vector < std::size_t > V__(CONST_POP_SIZE);
    std::iota(V__.begin( ), V__.end( ), 0); std::transform (V__.begin( ), V__.end( ), V__.begin( ),
            [ ](const auto &x)
    {
      return (x * 2);
    }
    ) ;
    std::vector < double > vcmfalpha__(CONST_CUTOFF + 1); std::vector <
        double > vcmfkappa__(CONST_CUTOFF + 1);      /*
        generatecmf § 4.5 */
    generatecmf(vcmfalpha__, vcmfkappa__);
    std::vector < double > errs__(2 * CONST_SAMPLE_SIZE);
    int r = CONST_EXPERIMENTS + 1;
    while (--r > 0) {      /*
          § 4.12 */
      getcompletesampletree(ancestry__, sl__, V__, vcmfalpha__);      /*
          § 4.15 */
      readcompletetree(sl__, errs__, ancestry__);
    }      /*
          return the estimates $\widehat{r}_i^N(n)$ summed over the experiments */
    std::for_each (errs__.begin( ), errs__.end( ), [ ](const auto &x)
    {
      std::cout ≪ x ≪ '\n';
    }
    ) ; }
```

This code is used in chunk 21.

4.17 the main module

the *main* function

21
```
/*
§ 4.1 */
⟨ includes 5 ⟩ /*
§ 4.2 */
⟨ constants 6 ⟩ /*
§ 4.3 */
⟨ rngs 7 ⟩ /*
§ 4.4 */
⟨ pmf 8 ⟩ /*
§ 4.5 */
⟨ generate cmf 9 ⟩ /*
§ 4.6 */
⟨ get one random number of potential offspring 10 ⟩ /*
§ 4.7 */
⟨ assign chromosomes to offspring 11 ⟩ /*
§ 4.8 */
⟨ add a new generation 12 ⟩ /*
§ 4.9 */
⟨ a random sample 13 ⟩ /*
§ 4.11 */
⟨ incomplete 15 ⟩ /*
§ 4.10 */
⟨ immediate ancestor 14 ⟩ /*
§ 4.12 */
⟨ one complete sample tree 16 ⟩ /*
§ 4.13 */
⟨ update the site-frequency spectrum 17 ⟩ /*
§ 4.14 */
⟨ update $\widehat{r}_i^N(n)$ 18 ⟩ /*
§ 4.15 */
⟨ read a complete tree 19 ⟩ /*
§ 4.16 */
⟨ go ahead — get $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ 20 ⟩
int main( )
{ /*
setup_rng § 4.3 */
setup_rng(static_cast⟨std::size_t⟩(atoi(argv[1]))); /*
estimate § 4.16 */
estimate( );
gsl_rng_free(rngtype);
return GSL_SUCCESS;
}
```

5 conclusions and bibliography

We estimate $\widehat{\varrho}_i^N(n) = \mathbb{E}\left[\mathbb{E}\left[R_i^N(n) : \mathcal{A}^{(N,n)}\right]\right]$ when the sample comes from a finite diploid panmictic population of constant size; the population may be evolving acccording to random recruitment success. To estimate $\widehat{\varrho}_i^N(n)$ we evolve the population forward in time and record the ancestry of the entire population; when a random sample with a complete tree is found the ancestry of the sample is fixed and we can read the branch lengths off the fixed complete tree; this process is repeated a given number of times and so an estimate of $\widehat{\varrho}_i^N(n)$ is obtained. The idea behind $\widehat{\varrho}_i^N(n)$ is the simple fact that the gene copies of a sample share an ancestry, a gene genealogy, even if the genealogy is unknown. One would want to compare $\widehat{\varrho}_i^N(n)$ to $\mathbb{E}\left[R_i^N(n)\right]$ (mean relative branch lengths evaluated without conditioning on the ancestry) to see the effect on gene genealogies of conditioning on complete sample trees.

References

[1] Birkner, M., Liu, H., and Sturm, A. Coalescent results for diploid exchangeable population models. *Electronic Journal of Probability 23*, none (Jan. 2018).

[2] Kernighan, B. W., and Ritchie, D. M. The C programming language, 1988.

[3] Knuth, D. E., and Levy, S. *The CWEB system of structured documentation: version 3.0*. Addison-Wesley Longman Publishing Co., Inc., Reading, Massachusetts, 1994.

[4] Schweinsberg, J. Coalescent processes obtained from supercritical Galton-Watson processes. *Stoch Proc Appl 106* (2003), 107–139.

[5] Tange, O. *GNU Parallel 20221122*, 2022. Zenodo.

Index

__ancestry: 19.
__ells: 17, 18.
__errs: 18.
__g: 19.
__m: 17.
__V: 16.
a: 8.
addgeneration: 12, 16.
ancestry: 15, 16.
ancestry__: 20.
argv: 21.
assert: 9, 10, 12, 13, 14, 15, 17, 18, 19.
assignchromstooffspring: 11, 12.
atoi: 21.
b: 15.
begin: 9, 12, 13, 15, 16, 17, 18, 19, 20.
ceil: 6.
clear: 12, 15, 16, 19.
cmfA: 9.
cmfalpha: 9.
CONST_A: 6, 9.
CONST_ALPHA: 6, 9.
CONST_CUTOFF: 6, 9, 20.
CONST_EXPERIMENTS: 6, 20.
CONST_N: 6, 12.
CONST_POP_SIZE: 6, 12, 14, 15, 16, 19, 20.
CONST_SAMPLE_SIZE: 6, 13, 15, 17, 19, 20.
CONST_VAREPSILON: 6.
copy: 13.
cout: 20.
d: 18.
dN: 6.
ells: 19.
empty: 15.
end: 9, 12, 13, 15, 16, 17, 18, 19, 20.
errs: 19.
errs__: 20.
estimate: 20, 21.
first: 12, 19.
for_each: 15, 17, 19, 20.
g: 14, 15.
generatecmf: 9, 20.
getagi: 14, 15, 19.
getcompletesampletree: 16, 20.
gsl_rng: 7.
gsl_rng_alloc: 7.
gsl_rng_default: 7.
gsl_rng_env_setup: 7.
gsl_rng_free: 21.
gsl_rng_set: 7.
gsl_rng_type: 7.
gsl_rng_uniform_pos: 10, 11.
GSL_SUCCESS: 21.
i: 9, 11, 12, 15.
insert: 15.
iota: 12, 16, 20.
j: 10.
k: 8, 11.
l: 14.
log: 6.
m: 19.
main: 21.
make_pair: 11.
mt19937_64: 7.
number_generation: 16.
pair: 11, 12.
pmf: 8, 9.
pow: 8.
push_back: 11, 12.
r: 20.
random_device: 7.
randomsample: 13, 16.
randomseed: 7.
randomX: 10, 12.
readcompletetree: 19, 20.
reserve: 12, 16.
resize: 15.
rng: 7, 12, 13.
rngtype: 7, 10, 11, 21.
s: 7, 12, 15.
sA: 9.
salpha: 9.
sample: 13, 15.
sampledlevels: 16.
second: 12, 17, 19.
setup_rng: 7, 21.
shuffle: 12, 13.
size: 12, 13, 14, 15, 19.
sl: 19.
sl__: 20.
std: 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21.
swap: 16.
T: 7.
tmp: 19.
transform: 9, 15, 18, 20.
tree: 12, 16.
treeincomplete: 15, 16.
u: 10.
unordered_map: 17, 19.
unordered_set: 15.
updaterrs: 18, 19.
updatesfs: 17, 19.
V__: 20.

vcmf: 12, 16.
vcmfalpha_ _: 20.
vcmfkappa_ _: 20.
vector: 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20.
x: 9, 11, 12, 17, 18, 19, 20.
y: 12, 15, 18.
z: 12.

List of Refinements

⟨ a random sample 13 ⟩ Used in chunk 21.
⟨ add a new generation 12 ⟩ Used in chunk 21.
⟨ assign chromosomes to offspring 11 ⟩ Used in chunk 21.
⟨ constants 6 ⟩ Used in chunk 21.
⟨ generate cmf 9 ⟩ Used in chunk 21.
⟨ get one random number of potential offspring 10 ⟩ Used in chunk 21.
⟨ go ahead — get $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ 20 ⟩ Used in chunk 21.
⟨ immediate ancestor 14 ⟩ Used in chunk 21.
⟨ includes 5 ⟩ Used in chunk 21.
⟨ incomplete 15 ⟩ Used in chunk 21.
⟨ one complete sample tree 16 ⟩ Used in chunk 21.
⟨ pmf 8 ⟩ Used in chunk 21.
⟨ read a complete tree 19 ⟩ Used in chunk 21.
⟨ rngs 7 ⟩ Used in chunk 21.
⟨ update $\widehat{r}_i^N(n)$ 18 ⟩ Used in chunk 21.
⟨ update the site-frequency spectrum 17 ⟩ Used in chunk 21.