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

Bjarki Eldon[1,2]

Let $\mathbb{N}_0 \equiv \{0,1,2,\ldots\}$ and $\left\{\xi^{n,N}\right\} \equiv \left\{\xi^{n,N}(g) : g \in \mathbb{N}_0\right\}$ be the ancestral process tracking the random ancestral relations of a sample of gene copies; $\tau^N(n) \equiv \inf\left\{g \in \mathbb{N}_0 : \#\xi^{n,N}(g) = 1\right\}$ and $L_i^N(n) \equiv \sum_{j=0}^{\tau^N(n)} \#\left\{\xi \in \xi^{n,N}(j) : \#\xi = i\right\}$ for $i = 1,2,\ldots,2n-1$, $L^N(n) \equiv \sum_{j=0}^{\tau^N(n)} \#\xi^{n,N}(j)$, where $\xi^{n,N}(0) \equiv \{\{1,2\},\ldots,\{2n-1,2n\}\}$. Then $R_i^N(n) \equiv L_i^N(n)/L^N(n)$ and $L^N(n) = \sum_j L_j^N(n)$. With this C++ code one estimates the functionals $\mathbb{E}\left[R_i^N(n)\right]$ of gene genealogies of samples from a finite diploid panmictic population of constant size absent selfing and evolving according to specific models of sweepstakes reproduction

Contents

1 Copyright

[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. October 25, 2025

2 Compilation, output and execution

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] 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 (GSL). 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 -x c++ 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 484433-83230401 -n1) > resout<i>
```

for $i = 1, \ldots, y$ and use `parallel`[Tan22]

```
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.

It follows from Definition 3.1, and emphasized in Remark 3.2, that two gene copies in the same diploid individual must have come from separate parents . It is thus necessary to track the pairing of marked gene copies (ancestral to the sampled gene copies) in diploid individuals.

Let $\{\xi^{n,N}(r) : r \in \mathbb{N}\cup\{0\}\}$ 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 $\tau^N(n) := \inf\{j \in \mathbb{N} : \#\xi^{n,N}(j) = 1\}$, and $\tau(n) := \inf\{t \geq 0 : \#\xi^n(t) = 1\}$. Consider the functionals

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

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

[BLS18]. The functionals $L_i^N(n)$ and $L_i(n)$ are the random length of branches supporting $i \in [n-1]$ leaves, $L^N(n) = L_1^N(n) + \cdots + L_{n-1}^N(n)$, and $L(n) = L_1(n) + \cdots + L_{n-1}(n)$.

Recall n is the number of diploid individuals sampled, so we have $2n$ gene copies. Let $L_i^N(n)$ denote the random length of branches supporting $i \in \{1,2,\ldots,2n-1\}$ leaves; write $R_i^N(n) \equiv$

$L_i^N(n)/\sum_{j=1}^{2n-1}(n)$. We estimate $\mathbb{E}\left[R_i^N(n)\right]$ when the sample comes from a finite haploid panmictic population evolving according to random recruitment success. By "recruitment success" we mean an increased chance of producing surviving offspring that number on the order of the population size.

Write $n = \{1, 2, \ldots, n\}$ for any $n \in \mathbb{N} := \{1, 2, \ldots\}$. 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

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

with $a > 0$ and where $C > 0$ is such that $\mathbb{P}(X \in [\zeta(N)]) = 1$. 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. The model in (1) is an extension of the one in [Sch03, Equation 11].

Let $0 < \alpha < 2$ and $\kappa \geq 2$ be fixed. We consider two models. Let E be the event

$$E \equiv \{X_i \rhd \mathbb{L}(\alpha, \zeta(N)), i = 1, 2, \ldots, N\}$$

and E^{c} where κ replaces α in E. The $X_1, \ldots, X_N$ are iid and

$$X_1 \rhd \mathbb{L}\left(\mathbb{1}_{\{E\}}\alpha + \mathbb{1}_{\{E^{\mathsf{c}}\}}\kappa, \zeta(N)\right) \tag{2}$$

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 algorithm is summarized in § 4; the code follows in § 4.1–§ 4.16; we conclude in § 5

4 code

We sample n diploid individuals and so $2n$ gene copies; the algorithm summarized :

1. $\left(r_1^N(n), \ldots, r_{2n-1}^N(n)\right) \leftarrow (0, \ldots, 0)$
2. For each of M experiments § 4.14
   (a) $\left(\ell_1^N(n), \ldots, \ell_{n-1}^N(n)\right) \leftarrow (0, \ldots, 0)$
   (b) initialise block sizes $((b_1, b_2), \ldots, (b_{2n-1}, b_{2n})) \leftarrow ((1,1), \ldots, (1,1))$
   (c) $m \leftarrow 2n$ the current number of ancestral gene copies
   (d) **while** $m > 1$:
      i. $\ell_{b_j}^N(n) \leftarrow 1 + \ell_{b_j}^N(n)$ for $b_j = b_1, \ldots, b_m$ § 4.12
      ii. sample numbers of potential offspring $X_1, \ldots, X_N$ § 4.7
      iii. given $X_1, \ldots, X_N$ assign marked diploid individuals to families § 4.8
      iv. merge blocks and append new blocks to sample § 4.10
   (e) given branch lengths $\ell_i^N(n)$ update estimate of $\mathbb{E}\left[R_i^N(n)\right]$ § 4.13

$$r_j^N(n) \leftarrow \frac{\ell_j^N(n)}{\sum_i \ell_i^N(n)} + r_j^N(n), \quad j = 1, \ldots, 2n-1$$

3. return an estimate $(1/M) r_i^N(n)$ of $\mathbb{E}\left[R_i^N(n)\right]$ § 4.15

4.1 includes

the included libraries; we use the GSL library so this needs to be installed

5 ⟨ includes 5 ⟩ ≡

```
#include <iostream>
#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 <forward_list>
#include <chrono>
#include <assert.h>
#include <math.h>
#include <unistd.h>
#include <omp.h>
#include <unordered_set>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_sf_log.h>
#include <gsl/gsl_sf_gamma.h>
```

This code is used in chunk 20.

4.2 constants

the parameter values

6 ⟨ constants 6 ⟩ ≡

```
const double CONST_ALPHA = 1.01;
const double CONST_A = 2.0;      /* ***
    N is number of parent pairs producing potential offspring; 2N is number of diploid
    individuals and so 4N gene copies in the population *** */
const std
    ::size_t CONST_N = 1 · 10^3;
    const double dN = static_cast⟨double⟩(CONST_N); const std
        ::size_t CONST_CUTOFF = 2 * CONST_N * static_cast⟨std::size_t⟩(ceil(log(dN)));
            const std::vector < double > P = {0.25, 0.5, 0.75, 1.};
        const double CONST_VAREPSILON = 0.1;      /*
            initialise number of diploid individuals sampled ; n is number of diploid
            individuals sampled so 2n is number of gene copies (blocks) *** */
        const std
            ::size_t CONST_SAMPLE_SIZE = 1 · 10^3;
            const int CONST_NUMBER_EXPERIMENTS = 2000;
```

This code is used in chunk 20.

4.3 the random number generators

the GSL random number generator

7 ⟨ rngs 7 ⟩ ≡

```
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 20.

4.4 the probability mass function for potential offspring

the probability mass function (1)

8 ⟨ pmf 8 ⟩ ≡

```
static double kernel(const double &k, const double &a)
{     /*
        compute k^{-a} - (1 + k)^{-a} */
  return (pow(1./k, a) - pow(1./(1. + k), a));
}
```

This code is used in chunk 20.

4.5 generate cmfs

generate cumulative mass functions

9 ⟨ cmfs 9 ⟩ ≡

```
static void generatecmf ( std::vector < double > &vcmfalpha, std::vector <
    double > &vcmfA ) {    /*
    the cmf arrays vcmfalpha and vcmfA are initialised to (0, ..., 0) *** */
double salpha
{ }
;
double sA
{ }
;
for (std::size_t i = 2; i < CONST_CUTOFF; ++i) {    /*
      kernel § 4.4 */
  vcmfalpha[i] = vcmfalpha[i − 1] + kernel(static_cast⟨double⟩(i), CONST_ALPHA);
  salpha += kernel(static_cast⟨double⟩(i), CONST_ALPHA);
  vcmfA[i] = vcmfA[i − 1] + kernel(static_cast⟨double⟩(i), CONST_A);
  sA += kernel(static_cast⟨double⟩(i), CONST_A);
}
assert(salpha > 0.);
assert(sA > 0.); std::transform (vcmfalpha.begin( ), vcmfalpha.end( ), vcmfalpha.begin( ),
        [&salpha](const auto &x)
{
  return x/salpha;
}
) ; std::transform (vcmfA.begin( ), vcmfA.end( ), vcmfA.begin( ), [&sA](const auto &x)
{
  return x/sA;
}
) ;
vcmfalpha[CONST_CUTOFF] = 1.;
vcmfA[CONST_CUTOFF] = 1.; }
```

This code is used in chunk 20.

4.6 sample one random number of potential offspring

sample one random number of potential offspring using (1); return $\min\{j : u \leq F(j)\}$ where F is a cumulative mass function and u a random uniform

10 ⟨ sample random X 10 ⟩ ≡

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

This code is used in chunk 20.

4.7 sample a pool of potential offspring

sample a pool of potential offspring using § 4.6

11 ⟨ sample pool potentials 11 ⟩ ≡

```
static std ::size_t sampleSN (std::vector < std::size_t > &vX, const std::vector <
        double > &f )
{    /*
       sample a pool of potential offspring */
  std::size_t sn
  {}
  ;
  for (std::size_t i = 0; i < CONST_N; ++i) {    /*
        randomX § 4.6 */
    vX[i] = randomX(f);
    sn += vX[i];
  }
  return sn;
}
```

This code is used in chunk 20.

4.8 sample number per family

Given a pool of potential offspring sample a random number of surviving marked diploid offspring assigned to a family; a 'marked' diploid offspring is one carrying a gene copy (chromosome) ancestral to a sampled chromosome

12 ⟨ random multivariate hypergeometric 12 ⟩ ≡

```
static void rmvhyper (std::vector < std::size_t > &merger_sizes, std::size_t &k, const
        std::vector < size_t > &v_juvs, const std ::size_t &SN )
{    /*
       k is the current number of marked individuals */
  merger_sizes.clear( );
  size_t n_others = SN − v_juvs[0];    /*
       sample the number of marked diploid potential offspring assigned to the first
       family */
  size_t x = gsl_ran_hypergeometric(rngtype, v_juvs[0], n_others, k);
  if (x > 0) {    /*
         only record merger sizes */
    merger_sizes.push_back(x);
  }    /*
         update the remaining number of blocks */
  k −= x;    /*
       update new number of lines */
  size_t i = 0;    /*
       we can stop as soon as all lines have been assigned to a family */
  while ((k > 0) ∧ (i < CONST_N − 1)) {    /*
         set the index to the one being sampled from */
    ++i;    /*
         update n_others */
    n_others −= v_juvs[i];
    x = gsl_ran_hypergeometric(rngtype, v_juvs[i], n_others, k);
    if (x > 0) {
      merger_sizes.push_back(x);
    }    /*
           update the remaining number of blocks */
    k −= x;
  }    /*
         check if at least two lines assigned to last individual */
  if (k > 0) {
    merger_sizes.push_back(k);
  }
}
```

This code is used in chunk 20.

4.9 sample parent chromosome

assigna a block to one of four parent chromosomes

13 ⟨ sample parent chromosome 13 ⟩ ≡

```
static std ::size_t samplepchrom( )
    {
      const double u = gsl_rng_uniform(rngtype);
      return (u < 0.25 ? 0 : (u < .5 ? 1 : (u < 0.75 ? 2 : 3)));
    }
```

This code is used in chunk 20.

4.10 merge blocks

merge blocks; blocks in the same diploid individual cannot merge, they disperse and have to be assigned to separate parents

14 ⟨ merge blocks 14 ⟩ ≡

```
static void mergeblocks ( const std ::size_t &j, std::vector < std::pair < std::size_t ,
        std::size_t ≫ &sample, const std ::size_t &x )
{    /*
       sample is a vector of pairs of block sizes; need to keep track of pairing of
       blocks in diploid individuals; blocks in same diploid individual disperse
       and so cannot merge */    /*
       x is number of marked diploid offspring of a given family */    /*
       newblocks will store the new block sizes of merged and/or continuing
       blocks */
  std::vector < std::size_t > newblocks(4, 0);
  assert(x > 0);    /*
       sample is a vector of marked diploid individuals */
  assert(j + x ≤ sample.size( ));
  for (std::size_t i = j; i < j + x; ++i) {    /*
         if marked diploid individual i is double-marked then the blocks disperse;
         otherwise use samplepchrom § 4.9 to sample parent chromosome */
    newblocks[sample[i].second > 0 ? (gsl_rng_uniform(rngtype) < 0.5 ? 0 : 1) :
         samplepchrom( )] += sample[i].first;
    newblocks[sample[i].first > 0 ? (gsl_rng_uniform(rngtype) < 0.5 ? 2 : 3) :
         samplepchrom( )] += sample[i].second;
  }    /*
         append the new blocks in pairs to the sample */
  if ((newblocks[0] + newblocks[1]) > 0) {
    sample.push_back(std::make_pair(newblocks[0], newblocks[1]));
  }
  if ((newblocks[2] + newblocks[3]) > 0) {
    sample.push_back(std::make_pair(newblocks[2], newblocks[3]));
  }
}
```

This code is used in chunk 20.

4.11 update sample

update sample by merging blocks and appending the new blocks to the sample

15 ⟨ update sample 15 ⟩ ≡

```
static std ::size_t updatesample ( const std ::size_t &offset, const std::vector
        < std::size_t > &vnu, std::vector < std::pair < std::size_t ,
        std::size_t ≫ &sample ) {      /*
        vnu is number of marked diploid offspring per parent pair */
    const std
        ::size_t newoffset = sample.size( );      /*
            o is the index of the first element to be worked with */
        std::size_t o = offset; for (const auto &m:vnu)
        {
          assert(m > 0);      /*
              § 4.10 */
          mergeblocks(o, sample, m);
          o += m;
        }     /*
              newoffset is the index of the first element of the new sample */
        return newoffset; }
```

This code is used in chunk 20.

4.12 update site-frequency spectrum $\ell_i^N(n)$

update the site-frequency spectrum (branch lengths $\ell_i^N(n)$); the sample is a vector of pairs of block sizes

16 ⟨ update branch lengths $\ell_i^N(n)$ 16 ⟩ ≡

```
static void updatebi (std::vector < std::size_t > &b, const std::vector <
    std::pair < std::size_t , std::size_t >> &s ) { for (const auto &p:s)
{
  assert(p.first < 2 * CONST_SAMPLE_SIZE);
  b[0] += (p.first > 0 ? 1 : 0);
  b[p.first] += (p.first > 0 ? 1 : 0);
  assert(p.second < 2 * CONST_SAMPLE_SIZE);
  b[0] += (p.second > 0 ? 1 : 0);
  b[p.second] += (p.second > 0 ? 1 : 0);
}
}
```

This code is used in chunk 20.

4.13 update estimate $r_i^N(n)$ of $\mathbb{E}\left[R_i^N(n)\right]$

given a realisation of branch lengths $\ell_i^N(n)$ update estimate $r_i^N(n)$ of $\mathbb{E}\left[R_i^N(n)\right]$

17 ⟨ update $r_i^N(n)$ 17 ⟩ ≡

```
static void updateri ( std::vector < double > &ri, const std::vector < std::size_t > &vb )
    {
      assert(vb[0] > 0);
      for (std::size_t i = 1; i < (2 * CONST_SAMPLE_SIZE); ++i) {
        ri[i] += (static_cast⟨double⟩(vb[i])/static_cast⟨double⟩(vb[0]));
      }
    }
```

This code is used in chunk 20.

4.14 one realisation of branch lengths

generate one realisation of branch lengths $\ell_i^N(n)$

18 ⟨ one branch length realisation 18 ⟩ ≡

```
static void one ( const std::vector < double > &vfalpha, const std::vector <
        double > &vfA, std::vector < std::size_t > &vbi, std::vector < double > &vri ) {
    /*
      initialise sample ((1,1),...,(1,1)) */
  std::vector < std::pair < std::size_t , std::size_t >> sample(CONST_SAMPLE_SIZE,
      std::make_pair(1,1));
  std::vector < std::size_t > vx(CONST_N,0); std::vector < std::size_t > vm
  {}
  ;
  vm.reserve(2 * CONST_SAMPLE_SIZE);
  std::size_t npo
  {}
  ;
  std::size_t ss
  {}
  ;
  std::size_t offs = 0;
  std::fill(std::begin(vbi), std::end(vbi), 0);
  while ((sample.back().first < 2 * CONST_SAMPLE_SIZE) ∧ (sample.back().second <
      2 * CONST_SAMPLE_SIZE)) {    /*
      updatebi § 4.12 */
    updatebi(vbi, sample);    /*
      sample potential offspring sampleSN § 4.7 */
    npo = sampleSN(vx, (gsl_rng_uniform(rngtype) < CONST_VAREPSILON ? vfalpha : vfA));
      /*
      sample number of marked potential offspring per family */    /*
      ss is number of marked diploid individuals */
    ss = sample.size() - offs;    /*
      rmvhyper § 4.8 */
    rmvhyper(vm, ss, vx, npo);
    assert(static_cast⟨std::size_t⟩(std::accumulate(vm.begin(), vm.end(),
      0)) ≡ (sample.size() - offs));    /*
      merge blocks and append new blocks to sample updatesample § 4.11 */
    offs = updatesample(offs, vm, sample);
  }    /*
      given realisation of branch lengths update estimate r_i^N(n) § 4.13 */
  updateri(vri, vbi); }
```

This code is used in chunk 20.

4.15 estimate $\mathbb{E}\left[R_i^N(n)\right]$

get estimates $r_i^N(n)$ of $\mathbb{E}\left[R_i^N(n)\right]$

19 ⟨ estimate $\mathbb{E}\left[R_i^N(n)\right]$ 19 ⟩ ≡

```
static void estimate( ) { std::vector < double > cmfalpha(CONST_CUTOFF + 1, 0.); std::vector
    < double > cmfA(CONST_CUTOFF + 1, 0.);      /*
    generatecmf § 4.5 */
  generatecmf(cmfalpha, cmfA);
  std::vector < std::size_t > branchlengths(2 * CONST_SAMPLE_SIZE, 0); std::vector <
    double > eri(2 * CONST_SAMPLE_SIZE, 0.);
  int r = CONST_NUMBER_EXPERIMENTS + 1;
  while (--r > 0) {      /*
      one § 4.14 */
    one(cmfalpha, cmfA, branchlengths, eri);
  }
  for (const auto &e:eri)
  {
    std::cout ≪ e ≪ '\n';
  }
  }
```

This code is used in chunk 20.

4.16 the main module

the *main* function

```
20      /*
        § 4.1 */
  ⟨ includes 5 ⟩    /*
        § 4.2 */
      ⟨ constants 6 ⟩    /*
        § 4.3 */
      ⟨ rngs 7 ⟩    /*
        § 4.4 */
      ⟨ pmf 8 ⟩    /*
        § 4.5 */
      ⟨ cmfs 9 ⟩    /*
        § 4.6 */
      ⟨ sample random X 10 ⟩    /*
        § 4.7 */
      ⟨ sample pool potentials 11 ⟩    /*
        § 4.8 */
      ⟨ random multivariate hypergeometric 12 ⟩    /*
        § 4.9 */
      ⟨ sample parent chromosome 13 ⟩    /*
        § 4.10 */
      ⟨ merge blocks 14 ⟩    /*
        § 4.11 */
      ⟨ update sample 15 ⟩    /*
        § 4.12 */
      ⟨ update branch lengths ℓ_i^N(n) 16 ⟩    /*
        § 4.13 */
      ⟨ update r_i^N(n) 17 ⟩    /*
        § 4.14 */
      ⟨ one branch length realisation 18 ⟩    /*
        § 4.15 */
      ⟨ estimate 𝔼[R_i^N(n)] 19 ⟩
      int main(int argc, const char *argv[])
      {    /*
          § 4.3 */
        setup_rng(static_cast⟨std::size_t⟩(atoi(argv[1])));    /*
          § 4.15 */
        estimate();
        gsl_rng_free(rngtype);
        return GSL_SUCCESS;
      }
```

5 conclusions and bibliography

We estimate mean relative branch lengths $\mathbb{E}\left[R_i^N(n)\right]$ for a sample from a finite diploid panmictic population of constant size (see Definition 3.1 and Remark 3.2) and evolving according to random recruitment success (1). Since the population is diploid each individual can carry two ancestral blocks; we exclude selfing (clonal reproduction) so blocks in the same diploid individual cannot merge; they disperse to the two parents.

One would want to compare $\mathbb{E}\left[R_i^N(n)\right]$ for some pre-limiting model to $\mathbb{E}\left[R_i(n)\right]$ predicted by the coalescent for which the pre-limiting model is in the domain-of-attraction to, in an effort to investigate how well the trees predicted by the two processes agree.

References

[BLS18] Matthias Birkner, Huili Liu, and Anja Sturm. Coalescent results for diploid exchangeable population models. *Electronic Journal of Probability*, 23(none), January 2018.

[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.

[Sch03] J Schweinsberg. Coalescent processes obtained from supercritical Galton-Watson processes. *Stoch Proc Appl*, 106:107–139, 2003.

[Tan22] O Tange. *GNU Parallel 20221122*, 2022. Zenodo.

Index

List of Refinements

⟨ cmfs 9 ⟩ Used in chunk 20.
⟨ constants 6 ⟩ Used in chunk 20.
⟨ estimate $\mathbb{E}\left[R_i^N(n)\right]$ 19 ⟩ Used in chunk 20.
⟨ includes 5 ⟩ Used in chunk 20.
⟨ merge blocks 14 ⟩ Used in chunk 20.
⟨ one branch length realisation 18 ⟩ Used in chunk 20.
⟨ pmf 8 ⟩ Used in chunk 20.
⟨ random multivariate hypergeometric 12 ⟩ Used in chunk 20.
⟨ rngs 7 ⟩ Used in chunk 20.
⟨ sample parent chromosome 13 ⟩ Used in chunk 20.
⟨ sample pool potentials 11 ⟩ Used in chunk 20.
⟨ sample random X 10 ⟩ Used in chunk 20.
⟨ update $r_i^N(n)$ 17 ⟩ Used in chunk 20.
⟨ update branch lengths $\ell_i^N(n)$ 16 ⟩ Used in chunk 20.
⟨ update sample 15 ⟩ Used in chunk 20.