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

Bjarki Eldon[1]

Write $[n] \equiv \{1,2,\ldots,n\}$ for $n \in \mathbb{N} \equiv \{1,2,\ldots\}$, $\mathbb{N}_0 \equiv \{0,1,2,\ldots\} = \mathbb{N} \cup \{0\}$. Let $\left\{\xi^{n,N}\right\} \equiv \left\{\xi^{n,N}(t), t \geq \mathbb{N}_0\right\}$ on the partitions of n denote the process keeping track of the random ancestral relations of a random set of individuals (gene copies) sampled at an arbitrary time from a haploid panmictic population of constant finite size N. The population may be evolving according to a model of sweepstakes reproduction (heavy-tailed offspring number distribution). Write $\#A$ for the number of elements in a given finite set A. Let $\tau^N(n) \equiv \inf\left\{t \in \mathbb{N}_0 : \#\xi^{n,N}(t) = 1\right\}$, $L_i^N(n) \equiv \sum_{t=0}^{\tau^N(n)} \#\left\{\xi \in \xi^{n,N}(t) : \#\xi = i\right\}$ for all $i \in [n-1]$, $L^N(n) \equiv \sum_{t=0}^{\tau^N(n)} \#\xi^{n,N}(t)$. Then $L^N(n) \equiv \sum_i L_i^N(n)$. Define $R_i^N(n) \equiv L_i^N(n)/L^N(n)$ for all $i \in [n-1]$. Interpreting $\left\{\xi^{n,N}\right\}$ as 'trees' we may view $R_i^N(n)$ as the random relative branch length supporting i leaves (sampled gene copies). With this C++ code we use simulations to approximate $\mathbb{E}\left[R_i^N(n)\right]$

Contents

[1] beldon11@gmail.com

compiled @ 10:44am on Wednesday 22nd October, 2025
6.16.12+deb14-amd64 GNU/Linux
CTANGLE 4.12.1 (TeX Live 2025/Debian)
g++ (Debian 15.2.0-4) 15.2.0
CWEAVE 4.12.1 (TeX Live 2025/Debian)
XƎLaTeX 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 Copyright

2 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.

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
```

See § 2.1 for a compilation script

2.1 a compilation script

Since the script below appears before the preamble one may use the shell tool `spix` on the script below; simply

`spix <name>.w`

```
1  NAFN=annealed_haploid_mixed_ERiN_wplotting
2  ctangle $NAFN.w
3  g++ -std=c++26 -m64 -march=native -O3 -x c++ $NAFN.c -lm -lgsl -lgslcblas
4  rm -f tmp_runs
5  for i in $(seq 4); do echo "./a.out 50 "  $(shuf -i 39393-282929191 -n1) ">
6  resout"$i >> tmp_runs; done
7  parallel --gnu -j4 :::: ./tmp_runs
8  paste resout* -d, P sed '1d' P awk -F, '{s=0; for (i=1;i<=NF;i++) {s+=$i} print
9  log(s/1e4) - log(1-(s/1e4))}' > logitresout
10 seq 49 P awk '{S=50;print log($1/S) - log(1 - ($1/S))}' > nlogits
11 paste -d',' nlogits logitresout > forplottingfile1
12 rm -f tmp_runs
13 for i in $(seq 4); do echo "./a.out 500 "  $(shuf -i 39393-282929191 -n1) ">
14 resout"$i >> tmp_runs; done
15 parallel --gnu -j4 :::: ./tmp_runs
16 paste resout* -d, P sed '1d' P awk -F, '{s=0; for (i=1;i<=NF;i++) {s+=$i} print
17 log(s/1e4) - log(1-(s/1e4))}' > logitresout
18 seq 499 P awk '{S=500;print log($1/S) - log(1 - ($1/S))}' > nlogits
19 paste -d',' nlogits logitresout > forplottingfile2
20 cweave $NAFN.w
21 tail -n4 $NAFN.tex > endi
22 for i in $(seq 5); do $(sed -i '$d' $NAFN.tex) ; done
23 cat endi >> $NAFN.tex
24 xelatex $NAFN.tex
```

In the script `P` stands for the shell pipe operator. For each of two parameter settings the simulations are run in parallel using `parallel`, we use `paste`, `sed`, and `awk` to process the results for plotting using the LaTeXpackage `tikz/pgf`; lines 21–23 in the script are just a hack to ensure a smooth compilation with XeTeX

One may also save the script in a file (say, compile) and apply `parallel` [Tan11]:

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

Figure 1 in § 5 graphs the results of the simulations generated by the script

3 intro

Recalling from the abstract let $\left\{\xi^{n,N}\right\} \equiv \left\{\xi^{n,N}(t), t \geq \mathbb{N}_0\right\}$ on the partitions of n denote the process keeping track of the random ancestral relations of a random set of individuals (gene copies) sampled at an arbitrary time from a haploid panmictic population of constant size N. The population may be evolving according to a model of sweepstakes reproduction (heavy-tailed offspring number distribution). Write $\#A$ for the number of elements in a given finite set A. Let $\tau^N(n) \equiv \inf\left\{t \in \mathbb{N}_0 : \#\xi^{n,N}(t) = 1\right\}$, $L_i^N(n) \equiv \sum_{t=0}^{\tau^N(n)} \#\left\{\xi \in \xi^{n,N}(t) : \#\xi = i\right\}$ for all $i \in [n-1]$, $L^N(n) \equiv \sum_{t=0}^{\tau^N(n)} \#\xi^{n,N}(t)$. Then $L^N(n) \equiv \sum_i L_i^N(n)$. Define $R_i^N(n) \equiv L_i^N(n)/L^N(n)$ for all $i \in [n-1]$. Interpreting $\left\{\xi^{n,N}\right\}$ as 'trees' we may view $R_i^N(n)$ as the random relative branch length supporting i leaves (sampled gene copies). We use simulations to estimate $\mathbb{E}\left[R_i^N(n)\right]$ for $i = 1, 2, \ldots, n-1$.

In each generation the current N individuals independently produce *potential* offspring (offspring that may survive to maturity) according to

$$\mathbb{P}(X = k) = C\left(k^{-a} - (1+k)^{-a}\right) \tag{1}$$

for $a > 0$ fixed and $C > 0$ chosen such that $\mathbb{P}(X \geq 1) = 1$. We may use a *finite* (fixed) upper bound $\zeta(N)$ on X, and choose C in (1) such that $\mathbb{P}(1 \leq X \leq \zeta(N)) = 1$.

Write $X \rhd L(\alpha, \zeta(N))$ when the law of X is as in (1) for some given a and $\zeta(N)$. Write

$$\begin{aligned} E &= \{X_i \rhd L(\alpha, \zeta(N)) \text{ for all } i \in [N]\} \\ E^{\mathsf{c}} &= \{X_i \rhd L(\kappa, \zeta(N)) \text{ for all } i \in [N]\} \end{aligned}$$

for some fixed $0 < \alpha < 2$ and $\kappa \geq 2$. Let $(\varepsilon_N)_{N \in \mathbb{N}}$ be a fixed positive sequence where $0 \leq \varepsilon_N \leq 1$ and it may hold that $\varepsilon_N \to 0$ as $N \to \infty$. Suppose

$$\mathbb{P}(E) = \varepsilon_N, \quad \mathbb{P}(E^{\mathsf{c}}) = 1 - \varepsilon_N \tag{2}$$

When $0 < \alpha < 1$ we take $\varepsilon_N = c(\log N)/N$.

In § 4.1 we summarize the algorithm, the code follows in § 4.2 – § 4.17, we conclude in §5. Comments within the code are in **this font and colour**

4 code

4.1 the algorithm

```
1. $(r_1, \dots, r_{n-1}) \leftarrow (0, \dots, 0)$
2. for each of $M$ experiments
    (a) $(\ell_1, \dots, \ell_{n-1}) \leftarrow (0, \dots, 0)$
    (b) $(\xi_1, \dots, \xi_n) \leftarrow (1, \dots, 1)$
    (c) $m \leftarrow n$ the current number of blocks
    (d) **while** $m > 1$ :
        i. $\ell_\xi \leftarrow \ell_\xi + 1$ for $\xi = \xi_1, \dots, \xi_m$
        ii. sample $N$ numbers of potential offspring $X_1, \dots, X_N$ (1)
        iii. get merger sizes $k_1, \dots, k_r$ by assigning blocks to families using $X_1, \dots, X_N$
        iv. merge blocks assigned to same family
        v. $m \leftarrow m - \sum_i \mathbb{1}_{\{k_i \geq 2\}} k_i + \sum_i \mathbb{1}_{\{k_i \geq 2\}}$
    (e) $r_i \leftarrow r_i + \ell_i / \sum_j \ell_j$ for $i = 1, 2, \dots, n-1$
3. return $r_i/M$ as an approximation of $\mathbb{E}\left[R_i^N(n)\right]$ for $i = 1, 2, \dots, n-1$
```

4.2 includes

the included libraries

7 ⟨includes 7⟩ ≡

```
#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 <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 22.

4.3 constants

the global constants

8 ⟨constants 8⟩ ≡ /*

```
      α in (2) */
  constexpr double CONST_ALPHA = 0.25;   /*
      κ in (2) */
  constexpr double KAPPA = 2.0;   /*
      population size N */
  constexpr size_t CONST_POP_SIZE = 5 · 10^2;   /*
      ε_N = cf(N) */
  constexpr double Cc = 1.;   /*
      ε_N in (2) */
  constexpr double EPSILON = Cc *
      log(static_cast⟨double⟩(CONST_POP_SIZE))/static_cast⟨double⟩(CONST_POP_SIZE);   /*
      ζ(N) in (1) */
  constexpr double CONST_CUTOFF = floor(pow(static_cast⟨double⟩(CONST_POP_SIZE),
      0.) * log(static_cast⟨double⟩(CONST_POP_SIZE)));   /*
      number of experiments */
  constexpr double CONST_NUMBER_EXPERIMENTS = 2500.;   /*
      number of experiments for approximating coalescence probabilities */
  constexpr std
      ::size_t NUMBER_EXPS = 10000;
```

This code is used in chunk 22.

4.4 the random number generators

the standard library and GSL random number generators

9 ⟨the rngs 9⟩ ≡

```
std::random_device 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 22.

4.5 the probability mass function (1)

compute $C\left(j^{-a} - (1+j)^{-a}\right)$ where $C = 1/\left(1 - (1+\zeta(N))^{-a}\right)$

10 $\langle$compute (1) 10$\rangle \equiv$

```
static double massfunction(const double &j)
{
  return ((pow(1.0/j, CONST_ALPHA) − pow(1.0/(1. + j),
      CONST_ALPHA))/(1. − pow(1./(CONST_CUTOFF + 1.), CONST_ALPHA)));
}
```

This code is used in chunk 22.

4.6 generate cumulative mass function of (1)

generate the cumulative mass function of (1); the mass function in (1) is monotone decreasing $\mathbb{P}(X = j+1) \le \mathbb{P}(X = j)$

11 ⟨cmf 11⟩ ≡

```
static void generate_cdf ( std::vector < double > &vcdf )
{
   vcdf.clear( );
   vcdf.push_back(0.0);      /*
       using massfunction in § 4.5   */
   for (double j = 1; j ≤ CONST_CUTOFF; ++j) {
      vcdf.push_back(vcdf.back( ) + massfunction(j));
   }
   assert(abs(vcdf.back( ) − 1.) < 0.999999);
}
```

This code is used in chunk 22.

4.7 sample one random number of potential offspring

sample the random number of potential offspring produced by one individual; use when there is an upper bound $\zeta(N)$, $\zeta(N) < \infty$

12 ⟨sample an X 12⟩ ≡

```
static size_t sample_juveniles ( const std::vector < double > &vcdf )
{     /*
         the CDF is generated in § 4.6  */
  size_t j = 1;
  const double u = gsl_rng_uniform(rngtype);
  while (vcdf[j] < u) {
    ++j;
  }
  return (j);
}
```

This code is used in chunk 22.

4.8 sample an X when $\zeta(N) = \infty$

sample one random number of potential offspring when $\zeta(N) = \infty$, return $\lfloor U^{-1/a} \rfloor$ where U a standard random uniform on $(0, 1]$

13 ⟨ X when unbounded 13 ⟩ ≡

```
static std ::size_t randomX(const double &a)
      {
         return static_cast⟨std::size_t⟩(floor(1./pow(gsl_rng_uniform_pos(rngtype), 1./a)));
      }
```

This code is used in chunk 22.

4.9 sample a pool of potential offspring

get a pool of potential offspring produced by the current N individuals

```
14  ⟨pool 14⟩ ≡
  static std ::size_t sample_pool_juveniles ( std::vector < size_t > &pool_juvs, const std::vector
              < double > &v_cdf ) {
          const double a = gsl_rng_uniform(rngtype) < EPSILON ? CONST_ALPHA : KAPPA;      /*
              s will be the total number sampled */
          std::size_t s = 0;      /*
              use randomX § 4.8 when unbounded; x = randomX(a);   */      /*
              use sample_juveniles § 4.7 when bounded; x = sample_juveniles(v_cdf); */
          std::transform (pool_juvs.begin( ), pool_juvs.end( ), pool_juvs.begin( ), [&s, &a](auto &x)
          {
            x = randomX(a);
            s += x;
            return x;
          }
          ) ;
          assert(s ≥ CONST_POP_SIZE);
          return (s); }
```

This code is used in chunk 22.

4.10 approximate coalescence probabilities

estimate coalescence probabilities for speeding up reaching the most recent common ancestor. When only two blocks left we can sample a geometric with success probability the pairwise coalescence probability. When only three blocks left can sample between a pairwise merger and a triple merger. Given a realisation $x_1, \ldots, x_N$ of $X_1, \ldots, X_N$ with $s_N := x_1 + \cdots + x_N$ the pairwise coalescence probability is

$$c_N = \sum_{j=1}^{N} \frac{x_j(x_j - 1)}{s_N(s_N - 1)}, \tag{3}$$

a 3-merger when three blocks is

$$c_N(3;3) = \sum_{j=1}^{N} \frac{(x_j)_3}{(s_N)_3}, \tag{4}$$

a 2-merger when three blocks is

$$c_N(3;2) = \sum_{j=1}^{N} \frac{3(x_j)_2(s_N - x_j)}{(s_N)_3}. \tag{5}$$

15 ⟨coalescence probs 15⟩ ≡

```
static void estimate_coalescence_probabilities ( std::vector < double > &v_cN, const
        std::vector < double > &v_cdf, std::vector < size_t > &v_pool_jvs )
    {
      size_t SN
      {}
      ;    /*
          NUMBER_EXPS § 4.3 */
      for (std::size_t i = 0; i < NUMBER_EXPS; ++i) {
        SN = sample_pool_juveniles(v_pool_jvs, v_cdf);
        for (size_t j = 0; j < CONST_POP_SIZE; ++j) {    /*
            (3) */
          v_cN[0] += (static_cast⟨double⟩(v_pool_jvs[j])/static_cast⟨double⟩(SN)) *
              (static_cast⟨double⟩(v_pool_jvs[j] - 1)/static_cast⟨double⟩(SN - 1));    /*
              (4) */
          v_cN[1] += (static_cast⟨double⟩(v_pool_jvs[j])/static_cast⟨double⟩(SN)) *
              (static_cast⟨double⟩(v_pool_jvs[j] - 1)/static_cast⟨double⟩(SN - 1)) *
              (static_cast⟨double⟩(v_pool_jvs[j] - 2)/static_cast⟨double⟩(SN - 2));    /*
              (5) */
          v_cN[2] += 3. * (static_cast⟨double⟩(SN - v_pool_jvs[j])/static_cast⟨double⟩(SN)) *
              (static_cast⟨double⟩(v_pool_jvs[j])/static_cast⟨double⟩(SN - 1)) *
              (static_cast⟨double⟩(v_pool_jvs[j] - 1)/static_cast⟨double⟩(SN - 2));
        }
      }
      v_cN[0] /= static_cast⟨double⟩(NUMBER_EXPS);
      v_cN[1] /= static_cast⟨double⟩(NUMBER_EXPS);
      v_cN[2] /= static_cast⟨double⟩(NUMBER_EXPS);
    }
```

This code is used in chunk 22.

4.11 assign blocks to families

assign blocks to families by sampling without replacement using given numbers of potential offspring

16 ⟨blocks to families 16⟩ ≡

```
static void rmvhyper ( std::vector < size_t > &merger_sizes, size_t k, const std::vector <
                size_t > &v_juvs, const size_t SN, gsl_rng*r )
      {    /*
              k is the current number of lines */
        merger_sizes.clear( );
        size_t n_others = SN − v_juvs[0];    /*
              sample the number of blocks assigned to the first family */
        size_t x = gsl_ran_hypergeometric(r, v_juvs[0], n_others, k);
        if (x > 1) {    /*
                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_POP_SIZE − 1)) {    /*
                set the index to the one being sampled from */
          ++i;    /*
                update n_others */
          n_others −= v_juvs[i];
          x = gsl_ran_hypergeometric(r, v_juvs[i], n_others, k);
          if (x > 1) {
            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 > 1) {
          merger_sizes.push_back(k);
        }
      }
```

This code is used in chunk 22.

4.12 merge blocks

17 ⟨block merging 17⟩ ≡

```
static void update_tree ( std::vector < size_t > &tree, const std::vector <
        size_t > &merger_sizes ) {
    std::vector < size_t > new_blocks
    { }
    ;
    if (merger_sizes.size( ) > 0) {      /*
         at least one merger */
    new_blocks.clear( );      /*
         shuffle the tree */
    std::ranges::shuffle(tree, rng);      /*
         loop over the mergers */
    for (const auto &m:merger_sizes)
    {      /*
           m is number of blocks merging; m is at least two; append new block to vector of
          new blocks */
      assert(m > 1);      /*
           record the size of the new block by summing the sizes of the merging blocks */
      new_blocks.push_back(std::accumulate(std::rbegin(tree), std::rbegin(tree) + m, 0));
      assert(new_blocks.back( ) > 1);      /*
           remove the rightmost m merged blocks from tree */
      tree.resize(tree.size( ) − m);
    }      /*
           append new blocks to tree */
    tree.insert(tree.end( ), new_blocks.begin( ), new_blocks.end( )); }      /*
         if no mergers then tree is unchanged */
    }
```

This code is used in chunk 22.

4.13 update $L_i^N(n)$

update lengths ℓ_i given current block sizes

18 ⟨update lengths 18⟩ ≡

```
static void update_ebib ( const std::vector < std::size_t > &tree, std::vector < double > &vebib
            ) { std::for_each (tree.begin( ), tree.end( ), [&vebib] ( const std ::size_t &i )
      {
         ++vebib[0];
         ++vebib[i];
      }
      ) ; }
```

This code is used in chunk 22.

4.14 update approximations of $\mathbb{E}\left[R_i^N(n)\right]$

given lengths ℓ_i from one tree update approximations r_i of $\mathbb{E}\left[R_i^N(n)\right]$

19 $\langle$update r_i 19$\rangle \equiv$

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

This code is used in chunk 22.

4.15 three or two blocks left

when at most three blocks left we use the coalescence probabilites (3), (4), (5) approximated in § 4.10

20 ⟨ 2 or 3 blocks 20 ⟩ ≡

```
  static void three_or_two_blocks_left ( std::vector < double > &tmp_bib, const std::vector <
        double > &v_cN, std::vector < size_t > &v_tree )
    {
      double Tk = 0.;
      double Tkk = 0.;
      size_t newblock
      { }
      ;
      switch (v_tree.size()) {
      case 3:
        {    /*
               three lines left so sample the two waiting times for a 3-merger and a 2-merger
             */
          Tk = static_cast⟨double⟩(gsl_ran_geometric(rngtype, v_cN[1]));
          Tkk = static_cast⟨double⟩(gsl_ran_geometric(rngtype, v_cN[2]));
          if (Tk < Tkk) {    /*
                all three blocks merge; update the branch lengths  */
            tmp_bib[0] += (3. * Tk);
            tmp_bib[v_tree[0]] += Tk;
            tmp_bib[v_tree[1]] += Tk;
            tmp_bib[v_tree[2]] += Tk;    /*
                clear the tree */
            v_tree.clear();
            assert(v_tree.size() < 1);
          }
          else {    /*
                a 2-merger occurs followed by a merger of the last two blocks */
            tmp_bib[0] += (3. * Tkk);
            tmp_bib[v_tree[0]] += Tkk;
            tmp_bib[v_tree[1]] += Tkk;
            tmp_bib[v_tree[2]] += Tkk;    /*
                shuffle the tree  */
            std::ranges::shuffle(v_tree, rng);
            newblock = v_tree[1] + v_tree[2];
            v_tree.resize(1);
            v_tree.push_back(newblock);
            assert(v_tree.size() ≡ 2);    /*
                sample waiting time until merger of last two blocks */
            Tk = static_cast⟨double⟩(gsl_ran_geometric(rngtype, v_cN[0]));
            tmp_bib[0] += (2. * Tk);
            tmp_bib[v_tree[0]] += Tk;
            tmp_bib[v_tree[1]] += Tk;
            v_tree.clear();
            assert(v_tree.size() < 1);
          }
          break;
        }
      case 2:
        {    /*
               two blocks left */
          Tk = static_cast⟨double⟩(gsl_ran_geometric(rngtype, v_cN[0]));
          tmp_bib[0] += (2. * Tk);
          tmp_bib[v_tree[0]] += Tk;
```

```
            $tmp\_bib[v\_tree[1]] += Tk$;
            $v\_tree.clear(\,)$;
            $assert(v\_tree.size(\,) < 1)$;
            **break**;
          }
        **default**: **break**;
        }
      }
```
This code is used in chunk 22.

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

approximate $\mathbb{E}\left[R_i^N(n)\right]$ given the settings in § 4.3

21 ⟨go ahead — approximate $\mathbb{E}\left[R_i^N(n)\right]$ 21⟩ ≡

```
  static void estimate_ebib ( const std ::size_t &n_leaves ) {      /*
            n_leaves is sample size n */
        std::vector < double > v_cdf(static_cast⟨std::size_t⟩(CONST_CUTOFF) + 1);      /*
              compute the CDF function for sampling juveniles §4.6 generate_cdf(v_cdf); */
        std::vector < size_t > v_number_juvs(CONST_POP_SIZE);      /*
              the tree; initially all blocks are singletons  */
        std::vector < size_t > v_tree(n_leaves, 1); std::vector < size_t > v_merger_sizes
        { }
        ;
        v_merger_sizes.reserve(n_leaves); std::vector < double > v_tmp_ebib(n_leaves, 0.0);
             std::vector < double > v_ebib(n_leaves, 0.0); std::vector < double > v_coal_probs(3,
             0.0);      /*
              estimate the coalescence probs §4.10 */
        estimate_coalescence_probabilities(v_coal_probs, v_cdf, v_number_juvs);
        size_t SN = 0;
        double number_experiments = CONST_NUMBER_EXPERIMENTS + 1.;
        while (−−number_experiments > 0.) {      /*
                initialise the tree as all singletons  */
          v_tree.clear( );
          v_tree.assign(n_leaves, 1);      /*
                initialise the container for the branch length for the current realisation  */
          std::fill(std::begin(v_tmp_ebib), std::end(v_tmp_ebib), 0.0);
          while (v_tree.size( ) > 1) {      /*
                  update ℓ_i §4.13 for the current tree configuration */
            update_ebib(v_tree, v_tmp_ebib);
            if (v_tree.size( ) > 3) {      /*
                    sample pool of juveniles § 4.9  */
              SN = sample_pool_juveniles(v_number_juvs, v_cdf);      /*
                    compute the merger sizes §4.11 */
              rmvhyper(v_merger_sizes, v_tree.size( ), v_number_juvs, SN, rngtype);      /*
                    update the tree §4.12 */
              update_tree(v_tree, v_merger_sizes);
            }
            else {      /*
                    at most three blocks left §4.15 */
              three_or_two_blocks_left(v_tmp_ebib, v_coal_probs, v_tree);
            }
          }      /*
                  update approximation of E[R_i^N(n)] §4.14  */
          update_estimate_ebib(v_tmp_ebib, v_ebib);
        }      /*
                print approximation of E[R_i^N]  */
        for (const auto &r:v_ebib)
        {
          std::cout ≪ r ≪ '\n';
        }
        }
```

This code is used in chunk 22.

4.17 main

the *main* module

22
```
        /*
        § 4.2 */
  ⟨ includes 7 ⟩     /*
            § 4.3 */
        ⟨ constants 8 ⟩     /*
            § 4.4 */
        ⟨ the rngs 9 ⟩     /*
            § 4.5 */
        ⟨ compute (1) 10 ⟩     /*
            § 4.6 */
        ⟨ cmf 11 ⟩     /*
            § 4.7 */
        ⟨ sample an X 12 ⟩     /*
            § 4.8 */
        ⟨ X when unbounded 13 ⟩     /*
            § 4.9 */
        ⟨ pool 14 ⟩     /*
            § 4.10 */
        ⟨ coalescence probs 15 ⟩     /*
            § 4.11 */
        ⟨ blocks to families 16 ⟩     /*
            § 4.12 */
        ⟨ block merging 17 ⟩     /*
            § 4.13 */
        ⟨ update lengths 18 ⟩     /*
            § 4.14 */
        ⟨ update r_i 19 ⟩     /*
            § 4.15 */
        ⟨ 2 or 3 blocks 20 ⟩     /*
            § 4.16 */
        ⟨ go ahead — approximate E[R_i^N(n)] 21 ⟩
        int main(int argc, char *argv[])
        {     /*
              §4.4 */
          setup_rng(static_cast⟨unsigned long int⟩(atoi(argv[2])));     /*
              §4.16 */
          estimate_ebib(static_cast⟨std::size_t⟩(atoi(argv[1])));
          gsl_rng_free(rngtype);
          return 0;
        }
```

5 conclusions and references

The idea behind approximating $\mathbb{E}\left[R_i^N(n)\right]$ is to compare $\mathbb{E}\left[R_i^N(n)\right]$ to $\mathbb{E}\left[R_i(n)\right]$, the quantities corresponding to $\mathbb{E}\left[R_i^N(n)\right]$ and predicted by $\{\xi^n\}$, the limit of $\left\{\xi^{n,N}\right\}$ as $N \to \infty$. For example, for $1 \le \alpha < 2$ the (time-rescaled) limit (in the sense of convergence of finite-dimensional distributions) of $\left\{\xi^{n,N}\right\}$ is the Beta$(2-\alpha,\alpha)$ coalescent [Sch03]. Here we provide the code for approximating $\mathbb{E}\left[R_i^N(n)\right]$ when the sample comes from a haploid panmictic population of constant size. The population may be evolving according to sweepstakes reproduction (heavy-tailed offspring number distribution; (1)).

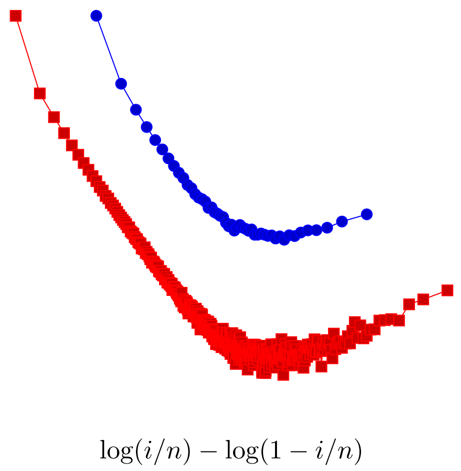

Figure 1: *An example approximation of* $\mathbb{E}\left[R_i^N(n)\right]$ *graphed as logits given the settings in § 4.3; see also § 2.1*

We use simulations to approximate $\mathbb{E}\left[R_i^N(n)\right]$; an example is given in Figure 1. One may (for example) investigate the effect of increasing n, or the upper bound $\zeta(N)$, on $\mathbb{E}\left[R_i^N(n)\right]$.

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

[Tan11] O Tange. GNU parallel – the command-line power tool. The USENIX Magazine, 2011.

Index

List of Refinements

$\langle$ X when unbounded 13 $\rangle$ Used in chunk 22.
$\langle$ 2 or 3 blocks 20 $\rangle$ Used in chunk 22.
$\langle$ block merging 17 $\rangle$ Used in chunk 22.
$\langle$ blocks to families 16 $\rangle$ Used in chunk 22.
$\langle$ cmf 11 $\rangle$ Used in chunk 22.
$\langle$ coalescence probs 15 $\rangle$ Used in chunk 22.
$\langle$ compute (1) 10 $\rangle$ Used in chunk 22.
$\langle$ constants 8 $\rangle$ Used in chunk 22.
$\langle$ go ahead — approximate $\mathbb{E}\left[R_i^N(n)\right]$ 21 $\rangle$ Used in chunk 22.
$\langle$ includes 7 $\rangle$ Used in chunk 22.
$\langle$ pool 14 $\rangle$ Used in chunk 22.
$\langle$ sample an X 12 $\rangle$ Used in chunk 22.
$\langle$ the rngs 9 $\rangle$ Used in chunk 22.
$\langle$ update r_i 19 $\rangle$ Used in chunk 22.
$\langle$ update lengths 18 $\rangle$ Used in chunk 22.