Beta-coalescents when sample size is large
— approximating $\mathbb{E}\left[R_i^N(n)\right]$

Bjarki Eldon[1]

Let $L_i^N(n) \equiv \sum_{j=1}^{\tau^N(n)} \#\left\{\xi \in \xi^{n,N}(j) : \#\xi = i\right\}$ and $L^N(n) \equiv \sum_{j=1}^{\tau^N(n)} \#\xi^{N,n}(j)$ and $\tau^N(n) \equiv \inf\left\{j \in \mathbb{N} : \#\xi^{n,N}(j) = 1\right\}$ for $i \in \{1, 2, \ldots, n-1\}$, and $R_i^N(n) \equiv L_i^N(n)/L^N(n)$ for $i = 1, 2, \ldots, n-1$ and $L^N(n) = \sum_j L_j^N(n)$. With this C++ code one estimates the functionals $\mathbb{E}\left[R_i^N(n)\right]$ for $i = 1, 2, \ldots, n-1$ of the ancestral process $\left\{\xi^{n,N}(r) : r \in \mathbb{N}_0\right\}$ tracking the random relations of n sampled gene copies when the sample is from a finite haploid panmictic population of constant size evolving according to a given model of sweepstakes reproduction (skewed offspring number distribution).

Contents

[1]Email: beldon11@gmail.com
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, WS, and BE; acknowledge Start-up module grants through SPP 1819 with Jere Koskela and Maite Wilke-Berenguer, and with Iulia Dahmer.
October 25, 2025

1 Copyright

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` 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++ -Wall -Wextra -pedantic -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 -show-leak-kinds=all <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` [5]

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

3 introduction

We consider a haploid population of fixed size N. Let $X, X_1, \ldots, X_N$ be i.i.d. discrete random variables taking values in $\{1, \ldots, \zeta(N)\}$; the $X_1, \ldots, X_N$ denote the random number of potential offspring independently produced in a given generation according to

$$\mathbb{P}(X = k) = \frac{(\zeta(N)+1)^\alpha}{(\zeta(N)+1)^\alpha - 1}\left(\frac{1}{k^\alpha} - \frac{1}{(k+1)^\alpha}\right), \quad 1 \leq k \leq \zeta(N). \tag{1}$$

The mass in (1) is normalised so that $\mathbb{P}(1 \leq X \leq \zeta(N)) = 1$, and $\mathbb{P}(X = k) \geq \mathbb{P}(X = k+1)$. Given a pool of at least N potential offspring, we sample N of them for the next generation uniformly at random and without replacement. Leaving out an atom at zero gives $X_1 + \cdots + X_N \geq N$ almost surely, guaranteeing that we always have at least N potential offspring to choose from in each generation. If $1 < \alpha < 2$ and $\liminf_{N\to\infty} \zeta(N)/N > 0$ the ancestral process tracing the random ancestral relations of leaves converges in finite-dimensional distributions to the Beta$(\gamma, 2-\alpha, \alpha)$-coalescent with $0 < \gamma \leq 1$; if $\alpha \geq 2$ or $\zeta(N)/N \to 0$ (the ancestral process) converges (in finite-dimensional distributions) to Kingman. Thus, the model described in (1) is a mathematically tractable model of sweepstakes reproduction (skewed offspring number distribution).

Let $\{\xi^{n,N}(j) : j \in \mathbb{N}_0\}$ where $\mathbb{N}_0 \equiv \{0, 1, 2, \ldots\}$ be the ancestral process tracking the random ancestral relations of sampled gene copies when the sample comes from a finite haploid panmictic population evolving according to (1). Let $L_i^N(n) \equiv \sum_{j=1}^{\tau^N(n)} \#\{\xi \in \xi^{n,N}(j) : \#\xi = i\}$ and $L^N(n) \equiv \sum_{j=1}^{\tau^N(n)} \#\xi^{N,n}(j)$ and $\tau^N(n) \equiv \inf\{j \in \mathbb{N} : \#\xi^{n,N}(j) = 1\}$ for $i \in \{1, 2, \ldots, n-1\}$, with n being the sample size. Then $L^N(n) = L_1^N(n) + \cdots + L_{n-1}^N(n)$; define $R_i^N(n) \equiv L_i^N(n)/L^N(n)$. We are interested in $\mathbb{E}\left[R_i^N(n)\right]$, and how $\mathbb{E}\left[R_i^N(n)\right]$ behaves as a function of n. This has been investigated in the case of evolution according to the Wright-Fisher model. We are interested in this question in the case of evolution according to sweepstakes reproduction, with numbers of potential offspring produced according to (1) and then sampled uniformly and without replacement.

The algorithm is summarised in § 4, the code follows in § 4.1–§ 4.16; we conclude in § 5. Comments within the code are in **this font and colour**

4 Code

The included libraries are listed in § 4.1, and the global constants in § 4.2. The random number generators are defined in § 4.3. The mass function in Eq (1) is computed in § 4.4, the corresponding CDF for sampling numbers of potential offspring is computed in § 4.5. A random number of potential offspring for a single individual is drawn in § 4.6, and for all the N individuals in § 4.8. From the pool of potential offspring (there are at least N of them almost surely) N are sampled without replacement, and this is the algorithm's Acciles's heel. Sampling without replacement is inefficient. To try to improve the efficiency in § 4.9 we estimate the pairwise coalescence probability c_N Eq (3). When there are two blocks left, the random time until the last two blocks merge is geometric with success probability c_N. When there are three blocks left, we can use a similar approach, i.e. estimate the corresponding coalescence probabilities Eq (4) and Eq (5), and sample geometric times In § 4.10 we assign blocks to families given a realisation of number of potential offspring per individual. If there are currently n blocks, the joint distribution of number of blocks per family is multivariate hypergeometric conditional on the number of potential offspring. Given a realisation $x_1 \dots, x_N$ of $X_1, \dots, X_N$ we approximate the number of blocks per family with

$$\mathbb{P}\left(\nu = (v_1, \dots, v_N)\right) = \frac{\binom{x_1}{v_1} \cdots \binom{x_N}{v_N}}{\binom{x_1 + \cdots + x_N}{n}} \tag{2}$$

This approximation well approximates first sampling N potential offspring and using the surviving offspring in the hypergeometric Eq (2). The hypergeometric step in the algorithm is the bottleneck. Given the number of blocks per family we can update the tree § 4.11. The tree is a vector of block sizes, and we merge blocks by first shuffling the order of the blocks, and then merging the rightmost blocks. The new blocks are then appended to the tree. Thus, we only store the current configuration of the tree. Obviously if at most one block is assigned to each family then the tree is unchanged over the generation. Given the current tree, the branch lengths for the current realisation are updated in § 4.12, and after each realisation the estimate of $\mathbb{E}\left[R_i^N(n)\right]$ is updated in § 4.13. We use the coalescence probability estimates § 4.9 in § 4.14, in the case when there are at most three blocks left in the tree.

Write $[n] = \{1, 2, \dots, n\}$ for any natural number n. Let n denote the sample size and $m \in [n]$ the current number of blocks. We are interested in the branch lengths and require the block sizes $(b_1, \dots, b_m)$ where $b_j \in [n]$ and $b_1 + \cdots + b_m = n$ are the current block sizes

1. $(r_1(n), \dots, r_{n-1}(n)) \leftarrow (0, \dots, 0)$
2. for each of M experiments :
   (a) set $(b_1, \dots, b_n) \leftarrow (1, \dots, 1)$.
   (b) set current branch lengths $\ell_i(n) \leftarrow 0$ for $i \in [n-1]$
   (c) set the current number of blocks $m \leftarrow n$
   (d) **while** $m > 1$ (at least two blocks; see § 4.15)
      i. update the current branch lengths $\ell_b(n) \leftarrow 1 + \ell_b(n)$ for $b = b_1, \dots, b_m$; § 4.12
      ii. sample random number of potential offspring $X_1, \dots, X_N$ § 4.8
      iii. given $X_1, \dots, X_N$ assign blocks to families § 4.10
      iv. merge blocks assigned to the same family and update current number of blocks § 4.11
   (e) given a realisation $\ell_1(n), \dots, \ell_{n-1}(n)$ of branch lengths update the estimate of $\mathbb{E}\left[R_i^N(n)\right]$ § 4.13 ; $r_i(n) \leftarrow r_i(n) + \ell_i(n) / \sum_j \ell_j(n)$

3. return an estimate $(1/M)r_i(n)$ of $\mathbb{E}\left[R_i^N(n)\right]$ for $i = 1, 2, \ldots, n-1$

4.1 includes

the included libraries; we use the GSL Library

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 <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 global constants

6 ⟨ constants 6 ⟩ ≡

```
/*
    the α parameter in (1) */
const double CONST_ALPHA = 1.0;    /*
    population size N */
const size_t CONST_POP_SIZE = 1 · 10^3;    /*
    the cutoff ζ(N) in (1) */
const double CONST_CUTOFF = 1.0 · 10^3;    /*
    sample size */
const size_t CONST_SAMPLE_SIZE = 10;    /*
    number of experiments */
const double CONST_NUMBER_EXPERIMENTS = 2500.;
```

This code is used in chunk 20.

4.3 the random number generator

7 ⟨ gslrng 7 ⟩ ≡

```
/*
    the GSL random number engine */
gsl_rng * rngtype;      /*
    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());      /*
    set up and initialise the GSL random number generator */
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 mass function

The mass function in (1)

8 ⟨ mass 8 ⟩ ≡

```
static double massfunction(const double j)
{    /*
       CONST_CUTOFF and CONST_ALPHA § 4.2; recall ζ(N) from (1)  */
  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 20.

4.5 cdf

define the function for computing the CDF for the distribution of number of potential offspring in (1).

9 ⟨ cdf 9 ⟩ ≡

```
static void mass_function ( std::vector < double > &vcdf )
{
  vcdf.clear( );      /*
      we take ℙ(X = 0) = 0 */
  vcdf.push_back(0.0);      /*
      CONST_CUTOFF § 4.2; recall ζ(N) from (1) */
  for (double j = 1; j ≤ CONST_CUTOFF; ++j) {      /*
        adding upp the mass function § 4.4 */
    vcdf.push_back(vcdf.back( ) + massfunction(j));
  }
}
```

This code is used in chunk 20.

4.6 random number of potential offspring

the function for sampling a random number of potential offspring returning $\min\{j \in \mathbb{N} : F(j) \geq u\}$ for u a random uniform, and F the CDF computed in § 4.5

10 ⟨ randomjuvs 10 ⟩ ≡

```
static size_t sample_juveniles ( const std::vector < double > &vcdf )
{
  size_t j = 1;
  const double u = gsl_rng_uniform(rngtype);
  while (vcdf[j] < u) {
    ++j;
  }
  return (j);
}
```

This code is used in chunk 20.

4.7 random unbounded potential offspring

unbounded; this is here for completeness more than anything else, one could interpret $\zeta(N) = N \log N$ as "unbounded" since then $\zeta(N)/N \to \infty$ as $N \to \infty$

11 ⟨ unbounded 11 ⟩ ≡

```
static std ::size_t randomX()
    {    /*
           CONST_ALPHA from § 4.2 */
      return floor(1./pow(gsl_rng_uniform_pos(rngtype), 1./CONST_ALPHA));
    }
```

This code is used in chunk 20.

4.8 pool of potential offspring

sample a random pool of potential offspring, i.e. each of the N individuals independently contributes a random number of potential offspring according to (1)

12 ⟨ pool 12 ⟩ ≡

```
static size_t sample_pool_juveniles ( std::vector < size_t > &pool_juvs, const std::vector
        < double > &v_cdf )
{
  pool_juvs.clear( );
  size_t s = 0;
  for (size_t i = 0; i < CONST_POP_SIZE; ++i) {      /*
        record the random number of potential offspring for each individual § 4.6
       */      /*
        sample_juveniles(v_cdf) for bounded */      /*
        randomX( ) for unbounded */
    pool_juvs.push_back(sample_juveniles(v_cdf));
    s += pool_juvs.back( );
  }      /*
        return the total number of juveniles X_1 + ··· + X_N   */
  return (s);
}
```

This code is used in chunk 20.

4.9 estimate 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

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

13 ⟨ estimatecoalpr 13 ⟩ ≡

```
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
  { }
  ;    /*
       estimate the coalescence probabilites from 10^3 experiments */
  for (size_t i = 0; i < 1000; ++i) {    /*
         sample a pool of potential offspring and record the total number of them
       § 4.8 */
    SN = sample_pool_juveniles(v_pool_jvs, v_cdf);
    for (size_t j = 0; j < CONST_POP_SIZE; ++j) {    /*
           the pairwise probability (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));
        /*
           a 3-merger when three blocks (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));
        /*
           a merger of two of three blocks Eq (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));
    }
  }    /*
         take the average */
```

```
    v_cN[0] /= 1000.;
    v_cN[1] /= 1000.;
    v_cN[2] /= 1000.;
  }
```

This code is used in chunk 20.

4.10 sample a multivariate hypergeometric

assign the ancestral blocks to families; given a realisation of random numbers of potential offspring the joint number of blocks per family is a multivariate hypergeometric. We sample the marginals and update.

14 ⟨ rmvhyper 14 ⟩ ≡

```
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 number_of_new_lines = 0;
  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 */
  number_of_new_lines += (x > 0 ? 1 : 0);
  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;   /*
      update new number of lines */
    number_of_new_lines += (x > 0 ? 1 : 0);
  }   /*
      check if at least two lines assigned to last individual */
  if (k > 1) {
    merger_sizes.push_back(k);
  }   /*
      check if at least one line assigned to last individual */
  if (k > 0) {
    number_of_new_lines += 1;
  }
}
```

This code is used in chunk 20.

4.11 update the tree

update the tree; the tree is a vector of block sizes. If there are mergers we shuffle the tree and then consequtively merge blocks by summing and recoding the size of the merging blocks in each merger, removing the blocks that merge and eventually adding the new blocks to the tree, the rightmost blocks merging each time.

15 ⟨ updatetree 15 ⟩ ≡

```
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 20.

4.12 update the branch lengths

update the branch lengths

16 ⟨ updateb 16 ⟩ ≡

```
static void update_ ebib ( const std::vector < size_t > &tree, std::vector < double > &vebib
    ) {
for (const auto &b:tree)
{    /*
       b is size of current block */    /*
       update the total tree size and then the branch length corresponding to the size of
      the block */
  vebib[0] += 1.0;
  vebib[b] += 1.0;
}
}
```

This code is used in chunk 20.

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

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

17 ⟨ updateri 17 ⟩ ≡

```
static void update_estimate_ebib ( const std::vector < double > &v_tmp, std::vector <
    double > &v_ebib )
{
  for (size_t i = 1; i < CONST_SAMPLE_SIZE; ++i) {
    v_ebib[i] += v_tmp[i]/v_tmp[0];
  }
}
```

This code is used in chunk 20.

4.14 three or two blocks left

at most three blocks left, so sample times using the estimates of the coalescence probabilities § 4.9

18 ⟨ threetwo 18 ⟩ ≡

```
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 20.

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

estimate $\mathbb{E}\left[R_i^N(n)\right]$ from a given number of experiments.

19 ⟨ theestimator 19 ⟩ ≡

```
static void estimate_ebib( ) {
    std::vector < double > v_cdf;
    v_cdf.reserve(static_cast⟨size_t⟩(CONST_CUTOFF) + 1);      /*
        compute the CDF function for sampling potential offspring § 4.5 */
    mass_function(v_cdf);
    std::vector < size_t > v_number_juvs;
    v_number_juvs.reserve(CONST_POP_SIZE);      /*
        the tree (current block sizes); initially all blocks are singletons (of size 1) */
    std::vector < size_t > v_tree(CONST_SAMPLE_SIZE, 1);
    std::vector < size_t > v_merger_sizes
    { }
    ;
    v_merger_sizes.reserve(CONST_SAMPLE_SIZE);
    std::vector < double > v_tmp_ebib(CONST_SAMPLE_SIZE, 0.0);
    std::vector < double > v_ebib(CONST_SAMPLE_SIZE, 0.0);
    std::vector < double > v_coal_probs(3, 0.0);      /*
        estimate the coalescence probs § 4.9 */
    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(CONST_SAMPLE_SIZE, 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);
      assert(std::accumulate(std::begin(v_tmp_ebib), std::end(v_tmp_ebib), 0.0) ≡ 0);
      while (v_tree.size( ) > 1) {      /*
          record the branch lengths for the current tree configuration */
        update_ebib(v_tree, v_tmp_ebib);
        if (v_tree.size( ) > 3) {      /*
            sample pool of potential offspring */
          SN = sample_pool_juveniles(v_number_juvs, v_cdf);      /*
              compute the merger sizes § 4.10 */
          rmvhyper(v_merger_sizes, v_tree.size( ), v_number_juvs, SN, rngtype);      /*
              update the tree § 4.11 */
          update_tree(v_tree, v_merger_sizes);
        }
        else {      /*
            at most three blocks left § 4.14 */
          three_or_two_blocks_left(v_tmp_ebib, v_coal_probs, v_tree);
        }
```

```
    }     /*
            update estimate of E[R_i^N(n)] */
    update_estimate_ebib(v_tmp_ebib, v_ebib);
  }     /*
          print the estimate of E[R_i^N] */
  for (const auto &r:v_ebib)
  {
    std::cout << r << '\n';
  }
  }
```

This code is used in chunk 20.

4.16 the main module

The *main* function

```
⟨ includes 5 ⟩
⟨ gslrng 7 ⟩
⟨ constants 6 ⟩
⟨ mass 8 ⟩
⟨ cdf 9 ⟩
⟨ randomjuvs 10 ⟩
⟨ unbounded 11 ⟩
⟨ pool 12 ⟩
⟨ estimatecoalpr 13 ⟩
⟨ rmvhyper 14 ⟩
⟨ updatetree 15 ⟩
⟨ updateb 16 ⟩
⟨ updateri 17 ⟩
⟨ threetwo 18 ⟩
⟨ theestimator 19 ⟩
int main(int argc, char *argv[])
{    /*
       initialise the GSL random number generator rngtype § 4.3 */
  setup_rng(static_cast⟨unsigned long int⟩(atoi(argv[1])));    /*
       estimate 𝔼[R_i^N(n)] § 4.15 */
  estimate_ebib();
  gsl_rng_free(rngtype);
  return GSL_SUCCESS;
}
```

5 conclusion and bibliography

In the Schweinsberg model [4] it is assumed that individuals can produce arbitrarily many potential offspring. From the pool of all potential offspring produced at any given time the surviving offspring are sampled without replacement (provided there is enough of them) to survive and replace the parents. In (1) an upper bound $\zeta(N)$ is introduced to the distribution of number of potential offspring. It can be shown that if $\zeta(N)/N \to K$ where $K > 0$ is fixed then the ancestral process converges to a Beta$(\gamma, 2-\alpha, \alpha)$-coalescent [1], a truncated (or incomplete) form of the Beta$(2-\alpha, \alpha)$-coalescent of [4]. With this C++ code we estimate the functionals $\mathbb{E}\left[R_i^N(n)\right]$; estimates of $\mathbb{E}\left[R_i^N(n)\right]$ may then be compared to estimates of $\mathbb{E}\left[R_i(n)\right]$, the functionals predicted by a given coalescent.

References

[1] JA Chetwyn-Diggle, Bjarki Eldon, and Matthias Hammer. Beta-coalescents when sample size is large. In preparation, 2025+.

[2] Brian W Kernighan and Dennis M Ritchie. The C programming language, 1988.

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

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

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

Index

List of Refinements

⟨ cdf 9 ⟩ Used in chunk 20.
⟨ constants 6 ⟩ Used in chunk 20.
⟨ estimatecoalpr 13 ⟩ Used in chunk 20.
⟨ gslrng 7 ⟩ Used in chunk 20.
⟨ includes 5 ⟩ Used in chunk 20.
⟨ mass 8 ⟩ Used in chunk 20.
⟨ pool 12 ⟩ Used in chunk 20.
⟨ randomjuvs 10 ⟩ Used in chunk 20.
⟨ rmvhyper 14 ⟩ Used in chunk 20.
⟨ theestimator 19 ⟩ Used in chunk 20.
⟨ threetwo 18 ⟩ Used in chunk 20.
⟨ unbounded 11 ⟩ Used in chunk 20.
⟨ updateb 16 ⟩ Used in chunk 20.
⟨ updateri 17 ⟩ Used in chunk 20.
⟨ updatetree 15 ⟩ Used in chunk 20.