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

Bjarki Eldon[1][2]

With this C++ code one estimates mean relative branch lengths $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ where $R_i^N(n) = L_i^N(n)/\sum_{j=1}^{n-1} L_j^N(n)$ and $L_i^N(n)$ is the random branch length supporting $i \in \{1, 2, \ldots, n-1\}$ leaves when the sample comes from a finite haploid panmictic population of constant size evolving according to sweepstakes reproduction (skewed offspring number distribution) and time is measured in discrete time steps (generations). The random sample of n leaves is from a finite haploid panmictic population of constant size N. The population evolves according to a model of sweepstakes reproduction. The key point is the conditioning on an ancestry (represented by the sigma-field $\mathcal{A}^{(N,n)}$ recording the relations between individuals); the population evolves forward in time and at each time step (generation) a random sample is drawn from the population and the sample tree checked for completeness; when a complete sample tree is obtained, i.e. when a common ancestor for all the leaves in the sample is found, the sample has a fixed ancestry, a fixed tree. The sample tree is traced and the branch lengths recorded. This process is then repeated a given number of times, each time starting from scratch with a new population, thus averaging over random ancestries (random complete sample trees). In this way one obtains an estimate of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ for $i = 1, 2, \ldots, n-1$.

Contents

[1] beldon11@gmail.com

[2] Funding by Icelandic Centre of Research (Rannís) through an Icelandic Research Fund (Rannsóknasjóður) Grant of Excellence no. 185151-051 to Einar Árnason, Katrín Halldórsdóttir, Wolfgang Stephan, Alison Etheridge, and BE; DFG SPP 1819 Programme Rapid Evolutionary Adaptation Start-up module grants with Jere Koskela, Maite Wilke Berenguer, and with Iulia Dahmer
October 25, 2025

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) 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 trixie/sid` with kernel `6.12.10-amd64` and `ctangle 4.11` and `g++ 14.2` and `GSL 2.8`

```
g++ -Wall -Wextra -pedantic -std=c++26 -O3 -march=native -m64 -x c++ <prefix>.c
-lm -lgsl -lgslcblas
```

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`(Tan11)

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

3 intro

A coalescent is a probabilistic description of the random ancestral relations of sampled gene copies (leaves). A coalescent $\{\xi\} \equiv \{\xi(t); t \geq 0\}$ is a Markov chain on the partitions of $\mathbb{N} = \{1, 2, \ldots\}$, where the only transitions are the merging of blocks (elements of a partition); restricting to $n \in N$ gives a coalescent $\{\xi^n(t) : t \geq 0\}$ on the partitions of $[n] = \{1, 2, \ldots, n\}$. We will consider coalescents describing the ancestral relations of gene copies of a single non-recombining locus (a contiguous non-recombining segment of a chromosome) in a single haploid panmictic population of constant size N. A "quenched coalescent" would be a coalescent obtained by conditioning on a random ancestry of the individuals in the population.

For the population model we consider an extension of the Schweinsberg model (Sch03). Let X denote the random number of potential offspring of an arbitrary individual. Then, with $1 < \alpha < 2$,

$$\mathbb{P}(X \geq k) = \frac{1}{k^\alpha}, \quad k \in \{1, 2, \ldots\} \tag{1}$$

is the unbounded distribution, and

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

with $\zeta(N)$ the upper bound on the distribution, with C so that $\mathbb{P}(1 \leq X \leq \zeta(N)) = 1$. The population evolves according to sweepstakes reproduction. In each generation the current individuals independently produce potential offspring according to (1) (or (2)), from the pool of all potential offspring N of them are sampled uniformly and without replacement to replace the current individuals (since $\mathbb{P}(X \geq 1) = 1$ we will have N potential offspring each and every time almost surely).

Let $L_i^N(n)$ denote the random total length of branches supporting $i \in \{1, 2, \ldots, n-1\}$ leaves; then $L^N(n) = L_1^N(n) + \cdots + L_{n-1}^N(n)$ is the total tree size, and $R_i^N(n) = L_i^N(n)/L^N(n)$ the relative branch length. The quantity $R_i^N(n)$ is well defined since $L^N(n) \geq n$ almost surely. We estimate $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ where the sigma-field $\mathcal{A}^{(N,n)}$ is the random ancestry of all the N individuals in the population and information on which n individuals (leaves) were sampled. The coalescence probability can be shown to be different between the annealed and the quenched coalescent when the two gene copies come from a population evolving according to a specific model of sweepstakes reproduction (DFBW24).

In Figure 1 in § 5 we compare estimates of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ to estimates of $\mathbb{E}[R_i(n)]$. It may not be a perfectly appropriate comparison but at this time we do not have a quenched coalescent. The comparison can still inform about how much $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ deviates from $\mathbb{E}[R_i(n)]$ as predicted by the annealed coalescent, e.g. as derived in (Sch03).

The algorithm is summarised in § 4, the code follows in § 4.2–§ 4.20; we conclude in § 6. Comments within the code are in **this font and colour**

4 code

The population tree is recorded as individuals living on levels, and each individual "points to" the level of its immediate ancestor.

generation :	levels
0:	1 2 3 4 5 6 7 8 9 10
1:	6 6 9 8 6 6 10 1 1 2
2:	4 5 10 9 6 1 5 10 10 2

Write $A_\ell(g)$ for the ancestor of individual on level ℓ in generation g. One may imagine the individuals are assigned levels, one individual on each level, and there are N levels. Suppose we sample the individual on level 2 in generation 2, then $A_2(2) = 5$, and $A_5(1) = 6$. Suppose we have also sampled the individual on level 6 in generation 2; then $A_6(2) = 1$, and $A_1(1) = 6$. The two sampled individuals share the ancestor on level 6 in generation 0. The ancestry of the two individuals is shown in red in the tree above for $N = 10$. The ancestry of the individuals on levels 8 and 9 in generation 2 is shown in blue. Each individual points to its immedate ancestor, and this is sufficient information to trace the ancestry of any given individual and find a common ancestor for any set of individuals.

4.1 a summary of the algorithm

In this section we summarize the algorithm for estimating $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$. We repeatedly simulate a population ancestry from scratch and regularly search for an ancestor of a random sample of leaves.

The algorithm may well be improved; the focus here is not on efficiency but correctness, to put together a working algorithm that does the correct thing.

1. initialise the population ancestry with the indexes $0, 1, \ldots, N-1$
2. draw uniformly without replacement n levels from $\{0, 1, 2, \ldots, N-1\}$ representing the levels of the leaves of a new sample § 4.10
3. until a complete sample tree is found § 4.12 :
   (a) sample a random number of potential offspring using § 4.8
   (b) record the surviving offspring each pointing to (or labelled with) its immediate ancestor and add the labels to the tree § 4.9
   (c) sample a new random sample § 4.10
4. when a complete sample tree is found the ancestry of the sample is fixed so trace the ancestry and record the branch lengths § 4.14, § 4.16
5. given the branch lengths for one sample update the estimate of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ § 4.17

4.2 Includes

The included libraries; we use the GSL library

6 ⟨ includes 6 ⟩ ≡

```
#include <iostream>
#include <fstream>
#include <vector>
#include <random>
#include <functional>
#include <memory>
#include <utility>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <cmath>
#include <list>
#include <string>
#include <fstream>
#include <chrono>
#include <forward_list>
#include <assert.h>
#include <math.h>
#include <unistd.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
```

This code is used in chunk 24.

4.3 Standard Library random number generator

define the standard library random number generator

7 ⟨ stdl rng 7 ⟩ ≡

```
/*
    obtain a seed out of thin air for the random number engine */
std::random_device randomseed;      /*
    Standard Mersenne twister random number engine */
std::mt19937_64 rng(randomseed());
```

This code is used in chunk 24.

4.4 the parameters

define the parameters of the model, the population size, the upper bound $\zeta(N)$ and α in (2), the sample size n and the number of experiments.

8 ⟨ parameters 8 ⟩ ≡

```
/*
    the population size N */
const std
    ::size_t CONST_POP_SIZE = 1 · 10^2;    /*
        the upper bound ζ(N) (2) */
    const std
        ::size_t CONST_CUTOFF = CONST_POP_SIZE;    /*
            α (1), (2) */
        const double CONST_ALPHA = 1.01;    /*
            sample size n */
        const std
            ::size_t CONST_SAMPLE_SIZE = 1 · 10^1;    /*
                number of experiments */
            const int CONST_EXPERIMENTS = 1 · 10^2;
```

This code is used in chunk 24.

4.5 the gsl random number generator

define the GSL random number generator *rngtype* and the initialising function

9 ⟨ gsl rng 9 ⟩ ≡

```
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 24.

4.6 the probability mass function (2)

compute the kernel $k^{-\alpha} - (1+k)^{-\alpha}$ of the mass function in (2)

10 ⟨ pmf 10 ⟩ ≡

```
static double kernel ( const std ::size_t &k )
    {
      return (pow(1./static_cast⟨double⟩(k),
          CONST_ALPHA) − pow(1./static_cast⟨double⟩(k + 1), CONST_ALPHA));
    }
```

This code is used in chunk 24.

4.7 generate the CMF for sampling potential offspring

generate the cumulative mass function (CMF) for sampling a random number of potential offspring according to (2)

11 ⟨ cmf 11 ⟩ ≡

```
static void generatecmf ( std::vector < double > &cmf ) {
double s
{ }
;
for (std::size_t i = 1; i ≤ CONST_CUTOFF; ++i) {      /*
        kernel § 4.6 */
  cmf[i] = cmf[i − 1] + kernel(i);
  s += kernel(i);
}
assert(s > 0.);      /*
      normalise to generate a probability distribution */
std::transform (cmf.begin( ), cmf.end( ), cmf.begin( ), [&s](const auto &x)
{
  return x/s;
}
) ;
cmf[CONST_CUTOFF] = 1.; }
```

This code is used in chunk 24.

4.8 sample a random number of potential offspring

sample a random number of potential offspring $\min\{j : u \leq F(j)\}$ where u is a random uniform and F the cumulative mass function § 4.7

12 ⟨ sample a number of offspring 12 ⟩ ≡

```
static std ::size_t randomX ( const std::vector < double > &f )
{    /*
       f is the cumulative mass function generated in § 4.7 */
  const double u = gsl_rng_uniform_pos(rngtype);
  std::size_t j
  {1};
  while (u > f[j]) {
    ++j;
  }
  assert(j ≥ 1);
  return j;
}
```

This code is used in chunk 24.

4.9 add a generation

add a generation to the population tree by sampling N surviving offspring among the potential offspring produced by the current individuals and recording the immediate ancestors of the new offspring

13 ⟨ add a generation 13 ⟩ ≡

```
static void addgeneration (std::vector < std::size_t > &tree, const std::vector <
    double > &vcmf ) {      /*
     y records the pool of potential offspring */
std::vector < std::size_t > y
{ }
;
y.clear( );
std::size_t x
{ }
;
for (std::size_t i = 0; i < CONST_POP_SIZE; ++i) {      /*
        randomX § 4.8 */
  x = randomX(vcmf);
  y.reserve(y.size( ) + x);      /*
       the individual on level i produces x potential offspring so in the pool of potential
      offspring there will be x offspring pointing to level i  */
  y.insert(y.end( ), x, i);
}
assert(y.size( ) ≥ CONST_POP_SIZE);      /*
     add new generation to tree */      /*
     y is the immediate ancestors of the new offspring */      /*
     shuffle the levels of the potential offspring and the first N will survive and be
    inserted into the tree, the population ancestry  */
std::shuffle(y.begin( ), y.end( ), rng);
tree.reserve(tree.size( ) + CONST_POP_SIZE);
std::move(y.begin( ), y.begin( ) + CONST_POP_SIZE, std::back_inserter(tree));
y.clear( );
y.shrink_to_fit( );
std::vector < std::size_t > ( ).swap(y); }
```

This code is used in chunk 24.

4.10 get a random sample

get a random sample by sampling labels uniformly at random without replacement; a sample with m blocks at time g is the vector $((s_1, \ell_1), \ldots, (s_m, \ell_m))$ where s_i is the size of block i and ℓ_i the level of the block; the immediate ancestor of the individual on level ℓ_i at time g is $A_{\ell_i}(g)$

14 ⟨ random sample 14 ⟩ ≡

```
static void randomsample(std::vector < std::pair < std::size_t, std::size_t >> &sample)
{    /*
       pair is (size of block, level of block) */    /*
       V will be the levels of the new sample */
  std::vector < std::size_t > V(CONST_POP_SIZE, 0);
  std::iota(V.begin( ), V.end( ), 0);
  std::shuffle(V.begin( ), V.end( ), rng);
  sample.clear( );
  assert(sample.size( ) < 1);
  sample.reserve(CONST_SAMPLE_SIZE);
  for (std::size_t i = 0; i < CONST_SAMPLE_SIZE; ++i) {    /*
         pair is (size of block, level of block) */    /*
         V[i] is the sampled level of leaf i */
    sample.push_back(std::make_pair(1, V[i]));
  }
  assert(sample.size( ) ≡ CONST_SAMPLE_SIZE);
}
```

This code is used in chunk 24.

4.11 the immediate ancestor

get the immediate ancestor (parent) $A_\ell(g)$ of the individual living on level ℓ at time g

15 ⟨ agi 15 ⟩ ≡

```
static std ::size_t getagi ( const std ::size_t &g, const std ::size_t &level, const
                             std::vector < std::size_t > &tree )
{     /*
        get A_level(g); */      /*
        the immediate ancestor of individual on level level at time g
       */
  return (tree[(g * CONST_POP_SIZE) + level]);
}
```

This code is used in chunk 24.

4.12 check if the sample tree is complete

check if the sample tree is complete with the leaves finding a common ancestor; here the sampled leaves go searching for a common ancestor

16 ⟨ coalesced 16 ⟩ ≡

```
static bool allcoalesced ( const std ::size_t &generations, const std::vector <
        std::pair < std::size_t , std::size_t ≫ &sample, const std::vector <
        std::size_t > &tre ) {      /*
        generations + 1 is the current number of generations in the tree (numbered
        from zero) */
    std::size_t g = generations;
    std::vector < std::size_t > a(CONST_SAMPLE_SIZE, 0);      /*
        pair is (size of block, level of block) */      /*
        copy sampled levels into a; use a to check if the tree is complete */
    std::transform (sample.begin( ), sample.end( ), a.begin( ), [](const auto &x)
    {
      return x.second;
    }
    ) ;
    while ((a.size( ) > 1) ∧ (g > 0)) {      /*
            record the immediate ancestors */
      for (std::size_t i = 0; i < a.size( ); ++i) {
        a[i] = getagi(g, a[i], tre);
      }      /*
              remove duplicate entries signalling common ancestors */
      std::sort(a.begin( ), a.end( ));
      a.erase(std::unique(a.begin( ), a.end( )), a.end( ));
      −−g;
    }      /*
            return TRUE if sample tree is complete */
    return (a.size( ) < 2); }
```

This code is used in chunk 24.

4.13 compare size of block

compare size of block for sorting in descending order on size of block

17 ⟨ compare 17 ⟩ ≡

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

This code is used in chunk 24.

4.14 update sample

update a sample given that the tree is complete; merge blocks and record continuing blocks

18 ⟨ update 18 ⟩ ≡

```
static void updatesample ( const std ::size_t &g, std::vector < std::pair < std::size_t ,
        std::size_t ≫ &sample, const std::vector < std::size_t > &tree )
{    /*
       pair is (size of block, level of block) */
  std::size_t s
  { }
  ;
  const std
      ::size_t e = sample.size( );
  std::size_t k
  { }
  ;
  for (std::size_t i = 0; i < e; ++i) {
    s = 0;
    for (std::size_t j = i; j < e; ++j) {
      s += (getagi(g, sample[i].second, tree) ≡ getagi(g, sample[j].second,
          tree) ? sample[j].first : 0);    /*
            if block has already been merged set size of block to zero */
      sample[j].first = (getagi(g, sample[i].second, tree) ≡ getagi(g,
          sample[j].second, tree) ? 0 : sample[j].first);
    }
    if (s > 0) {    /*
            record only a new merged block or a current block not merging */
      ++k;    /*
            record the continuing block with the block size */    /*
            and the level of the immediate ancestor of the block */
      sample.push_back(std::make_pair(s, getagi(g, sample[i].second, tree)));
    }
  }    /*
         sort the sample on block size in descending order using § 4.13 */
  std::sort(sample.begin( ), sample.end( ), comp);    /*
        remove all blocks with block size zero */
  sample.resize(k);
  assert(sample.back( ).first > 0);
  assert(sample.size( ) > 1 ? (sample.back( ).first < CONST_SAMPLE_SIZE) :
      (sample.back( ).first ≡ CONST_SAMPLE_SIZE));
}
```

This code is used in chunk 24.

4.15 update the current spectrum

update the current branch length spectrum

19 ⟨ current bls 19 ⟩ ≡

```
static void updatevbi ( const std::vector < std::pair < std::size_t ,
    std::size_t >> &sample, std::vector < std::size_t > &vbi )
{   /*
      pair is (size of block, level of block) */
  for (std::size_t i = 0; i < sample.size( ); ++i) {   /*
        update the total tree length */
    ++vbi[0];
    assert(sample[i].first > 0);
    assert(sample[i].first < CONST_SAMPLE_SIZE);   /*
        update ℓ_j^N(n) where j = sample[i].first */
    ++vbi[sample[i].first];
  }
}
```

This code is used in chunk 24.

4.16 record one branch length spectrum

record one realisation of $\big(L_1^N(n), \ldots, L_{n-1}^N(n)\big)$; the sample tree is complete so read the branch lengths off the tree

20 ⟨ one bls 20 ⟩ ≡

```
static void recordonebls (std::vector < std::pair < std::size_t, std::size_t >> &csample,
    const std::vector < std::size_t > &tre, std::vector < std::size_t > &bls )
{
  std::size_t g = (tre.size( )/CONST_POP_SIZE) − 1;
  std::fill(bls.begin( ), bls.end( ), 0);
  while (csample.back( ).first < CONST_SAMPLE_SIZE) {      /*
        § 4.15 for updatevbi */
    updatevbi(csample, bls);      /*
        § 4.14 for updatesample */
    updatesample(g, csample, tre);
    −−g;
  }
}
```

This code is used in chunk 24.

4.17 update the estimate $\overline{\rho}_i^N(n)$ of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$

21 $\langle\, \overline{\rho}_i^N(n) \text{ update } 21 \,\rangle \equiv$

```
static void updaterestimate ( const std::vector < std::size_t > &b, std::vector <
    double > &r )
{   /*
      b[0] is total tree size */
  assert(b[0] ≥ CONST_SAMPLE_SIZE);
  for (std::size_t i = 1; i < CONST_SAMPLE_SIZE; ++i) {
    r[i] += static_cast⟨double⟩(b[i])/static_cast⟨double⟩(b[0]);
  }
}
```

This code is used in chunk 24.

4.18 one ancestry

record one ancestry and the resulting branch lengths; add to the population tree and draw samples until a sample tree is complete, then trace the ancestry of the sample and record the branch lengths

22 ⟨ ancestry 22 ⟩ ≡

```
static void oneancestry ( std::vector < double > &vr, const std::vector < double > &vf
    ) {
  std::vector < std::size_t > tre(CONST_POP_SIZE, 0);
  std::iota(tre.begin( ), tre.end( ), 0); std::vector < std::pair < std::size_t , std::size_t >>
      v
  { }
  ;
  std::vector < std::size_t > b(CONST_SAMPLE_SIZE, 0);      /*
      § 4.10 for randomsample */
  randomsample(v);
  std::size_t numbergenerations = 0;      /*
      § 4.12 for allcoalesced */
  while (¬allcoalesced(numbergenerations, v, tre)) {      /*
      § 4.9 */
    addgeneration(tre, vf);
    randomsample(v);
    ++numbergenerations;
  }      /*
      sample merges so record the resulting bls; § 4.16 for recordonebls */
  recordonebls(v, tre, b);      /*
      update estimate of relative branch lengths; § 4.17 for updaterestimate */
  updaterestimate(b, vr); }
```

This code is used in chunk 24.

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

estimate $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ by generating `CONST_EXPERIMENTS` number of ancestries and tracing each for the branch lengths; recall § 4.4 for the parameter values

23 ⟨ estimate 23 ⟩ ≡

```
static void estimate( ) {      /*
        estimate the relative branch lengths */
    std::vector < double > vcmfx(CONST_CUTOFF + 1, 0); std::vector <
        double > vR(CONST_SAMPLE_SIZE, 0);      /*
        § 4.7 for generatecmf */
    generatecmf(vcmfx);
    int r = CONST_EXPERIMENTS + 1;
    while (--r > 0) {      /*
        § 4.18 for oneancestry */
      oneancestry(vR, vcmfx);
    }      /*
        record the estimate ρ̄_i^N(n) of E[R̃_i^N(n)] */
    for (const auto &z:vR)
    {
      std::cout ≪ z/static_cast⟨double⟩(CONST_EXPERIMENTS) ≪ '\n';
    }
}
```

This code is used in chunk 24.

4.20 the main module

The *main* function

```
24      /*
        § 4.2 */
    ⟨ includes 6 ⟩   /*
        § 4.3 */
    ⟨ stdl rng 7 ⟩   /*
        § 4.4 */
    ⟨ parameters 8 ⟩   /*
        § 4.5 */
    ⟨ gsl rng 9 ⟩   /*
        § 4.6 */
    ⟨ pmf 10 ⟩   /*
        § 4.7 */
    ⟨ cmf 11 ⟩   /*
        § 4.8 */
    ⟨ sample a number of offspring 12 ⟩   /*
        § 4.9 */
    ⟨ add a generation 13 ⟩   /*
        § 4.10 */
    ⟨ random sample 14 ⟩   /*
        § 4.11 */
    ⟨ agi 15 ⟩   /*
        § 4.12 */
    ⟨ coalesced 16 ⟩   /*
        § 4.13 */
    ⟨ compare 17 ⟩   /*
        § 4.14 */
    ⟨ update 18 ⟩   /*
        § 4.15 */
    ⟨ current bls 19 ⟩   /*
        § 4.16 */
    ⟨ one bls 20 ⟩   /*
        § 4.17 */
    ⟨ $\overline{\rho}_i^N(n)$ update 21 ⟩   /*
        § 4.18 */
    ⟨ ancestry 22 ⟩   /*
        § 4.19 */
    ⟨ estimate 23 ⟩
    int main(int argc, const char *argv[])
    {   /*
            § 4.5 for setup_rng */
      setup_rng(static_cast⟨unsigned long⟩(atoi(argv[1])));   /*
            § 4.19 for estimate */
      estimate();
      gsl_rng_free(rngtype);
      return GSL_SUCCESS;
    }
```

5 examples

Let the quenched (N, n)-coalescent denote the trees generated by evolving a haploid population forward in time until a random sample of n leaves coalesces (finds a most recent common ancestor); the sample then has a fixed tree or ancestry (recall § 3). Since we do not at this time have a quenched coalescent as $N \to \infty$ we compare the normalised branch lengths $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ predicted by the quenched (N, n)-coalescent to $\mathbb{E}\left[R_i(n)\right]$ predicted by the annealed coalescent obtained in the usual way as $N \to \infty$.

In Fig 1 we compare the branch length spectrum $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ predicted by the quenched (N, n)-coalescent to $\mathbb{E}\left[R_i(n)\right]$ predicted by the annealed incomplete n-Beta-coalescent with coalescent rates when $k = 2, 3, \ldots, n$

$$\lambda_{n,k} = \frac{1}{B(\gamma; 2-\alpha, \alpha)} \int_0^1 \mathbb{1}_{\{0<x\leq\gamma\}} x^{k-\alpha-1}(1-x)^{n+\alpha-k-1} dx \tag{3}$$

where $B(\gamma; 2-\alpha, \alpha) = \int_0^1 \mathbb{1}_{\{0<x\leq\gamma\}} x^{1-\alpha}(1-x)^{\alpha-1} dx$ and

$$\gamma = \frac{K}{K + m_\infty} \tag{4}$$

when $\zeta(N) = KN$ recall (2) for some constant $K > 0$, and $m_\infty = \lim_{N\to\infty} \mathbb{E}\left[X_1\right]$ the expected number of potential offspring in an arbitrarily large population; we approximate m_∞ with

$$m_\infty \approx \frac{1}{2}\left(2 + \frac{1 + 2^{1-\alpha}}{\alpha - 1}\right) \tag{5}$$

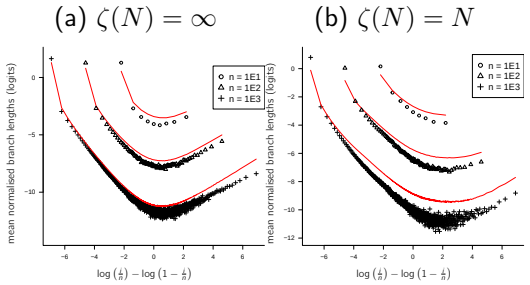

Figure 1: Comparing $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ and $\mathbb{E}\left[R_i(n)\right]$. Relative branch lengths compared between the quenched coalescent (symbols) for $N = 10^3$, $\alpha = 1.05$, $\zeta(N) = \infty$ according to (1) (a) $\zeta(N) = N$ according to (2) (b) compared to the BLS predicted by the annealed complete (a) and incomplete (b) Beta-coalescent (3) (red lines) for sample size n as shown with γ as in (4) where m_∞ as in (5) , recall we take $\zeta(N) = N$ for the quenched coalescent. Here we graph $\log(r_i(n)) - \log(1 - r_i(n))$ as a function of $\log(i/n) - \log(1 - i/n)$ for $i = 1, 2, \ldots, n-1$ where $r_i(n)$ is an estimate of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ (symbols) and $\mathbb{E}\left[R_i(n)\right]$ (red lines) recall the notation from § 3; the estimates of $\mathbb{E}\left[\widetilde{R}_i^N(n)\right]$ resp. $\mathbb{E}\left[R_i(n)\right]$ from 10^4 resp. 10^6 experiments

6 conclusions and bibliography

Write $(x)_m = x(x-1)\cdots(x-m+1)$ for any real x and $m \in \mathbb{N}$ and $(x)_0 \equiv 1$. Let $\nu_1, \ldots, \nu_N$ denote the random number of offspring of individuals in a haploid panmictic population of constant size N current in an arbitrary generation, $\sum_i \nu_i = N$. Let $c_N \equiv \mathbb{E}\left[\nu_1(\nu_1 - 1)\right]/(N-1)$. If $c_N \to 0$ and

$$\lim_{N\to\infty} \frac{\mathbb{E}\left[(\nu_1)_{k_1} \cdots (\nu_r)_{k_r}\right]}{N^{k_1+\cdots+k_r-r} c_N} \tag{6}$$

exist for all $k_1, \ldots, k_r \geq 2$ and $r \in \mathbb{N}$ then $\left\{\xi^{n,N}\left(\lfloor t/c_N \rfloor\right); t \geq 0\right\}$ converge to $\{\xi^n(t); t \geq 0\}$ (in finite-dimensional distributions) and the transition rates of $\{\xi^n(t); t \geq 0\}$ are uniquely determined by the limit in (6) (Sch03, Proposition 1) and (MS01). However, (6) implicitly averages over the ancestral relations of the gene copies in a sample.

With the C++ code presented here one can use simulations to check if conditioning on the (random) population ancestry matters for predictions of genetic variation, and thus for inference of evolutionary histories of natural populations.

References

[DFBW24] Dimitrios Diamantidis, Wai-Tong (Louis) Fan, Matthias Birkner, and John Wakeley. Bursts of coalescence within population pedigrees whenever big families occur. *GENETICS*, 227(1), February 2024.

[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.

[MS01] M Möhle and S Sagitov. A classification of coalescent processes for haploid exchangeable population models. *Ann Probab*, 29:1547–1562, 2001.

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

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

Index

List of Refinements

⟨ $\overline{\rho}_i^N(n)$ update 21 ⟩ Used in chunk 24.
⟨ add a generation 13 ⟩ Used in chunk 24.
⟨ agi 15 ⟩ Used in chunk 24.
⟨ ancestry 22 ⟩ Used in chunk 24.
⟨ cmf 11 ⟩ Used in chunk 24.
⟨ coalesced 16 ⟩ Used in chunk 24.
⟨ compare 17 ⟩ Used in chunk 24.
⟨ current bls 19 ⟩ Used in chunk 24.
⟨ estimate 23 ⟩ Used in chunk 24.
⟨ gsl rng 9 ⟩ Used in chunk 24.
⟨ includes 6 ⟩ Used in chunk 24.
⟨ one bls 20 ⟩ Used in chunk 24.
⟨ parameters 8 ⟩ Used in chunk 24.
⟨ pmf 10 ⟩ Used in chunk 24.
⟨ random sample 14 ⟩ Used in chunk 24.
⟨ sample a number of offspring 12 ⟩ Used in chunk 24.
⟨ stdl rng 7 ⟩ Used in chunk 24.
⟨ update 18 ⟩ Used in chunk 24.