Gene genealogies in haploid populations evolving according to sweepstakes reproduction — estimating $\mathbb{E}[R_i(n)]$ for the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent

Bjarki Eldon[1] [2]

Let $\{\xi^n(t) : t \geq 0\}$ be the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent, a Markov chain tracking the random relations of sampled gene copies, $\#A$ is the cardinality of a given set A, $L_i(n) \equiv \int_0^{\tau(n)} \#\{\xi \in \xi^n(t) : \#\xi = i\}\, dt$ and $L(n) \equiv \int_0^{\tau(n)} \#\xi^n(t) dt$ and $\tau(n) \equiv \inf\{t \geq 0 : \#\xi^n(t) = 1\}$ for $i \in \{1, 2, \ldots, n-1\}$; $R_i(n) \equiv L_i(n)/L(n)$ for $i = 1, 2, \ldots, n-1$. Then $L_i(n)$ is interpreted as the random total length of branches supporting $i \in \{1, 2, \ldots, n-1\}$ leaves, with the length measured in coalescent time units, and n sample size. We then have $L(n) = L_1(n) + \cdots + L_{n-1}(n)$. With this C++ code one estimates $\mathbb{E}[R_i(n)]$ predicted by the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent for $0 < \gamma \leq 1$ and $0 < \alpha < 2$. This family of multiple-merger coalescents can be shown to describe the gene genealogies of a sample of n gene copies from a haploid panmictic population of constant size evolving in a random environment (Definitions 3.2 and 3.3) according to sweepstakes reproduction [Eld24].

Contents

[1] beldon11@gmail.com

[2] compiled @ 10:44am on Tuesday 6th January, 2026
6.17.13+deb14-amd64 GNU/Linux
CTANGLE 4.12.1 (TeX Live 2025/Debian)
g++ (Debian 15.2.0-12) 15.2.0
GSL 2.8
CWEAVE 4.12.1 (TeX Live 2025/Debian)
This is LuaHBTeX, Version 1.22.0 (TeX Live 2025/Debian) Development id: 7673
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 30.2

1 Copyright

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.

Compiles on Linux debian 6.12.6-amd64 with CTANGLE 4.11 (TeX Live 2025/dev/Debian) and g++ 14.2 and GSL 2.8

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. 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 -vgdb=full -leak-resolution=high
-num-callers=50 <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

Suppose the population evolves as in Definition 3.1.

Definition 3.1 (Evolution of a haploid panmictic population) *Consider a haploid, i.e. each individual carries exactly one gene copy, panmictic population of constant size N evolving in discrete (non-overlapping) generations. In any given generation each individual independently produces a random number of potential offspring according to some given law. If the total number of potential offspring produced in this way is at least N, then N of them sampled uniformly at random without replacement survive to maturity and form a new set of reproducing individuals, and the remaining potential offspring perish (before reaching maturity). Otherwise we will assume an unchanged population over the generation (all the potential offspring perish before reaching maturity).*

Let $\left\{\xi^{N,n}(t); t \geq 0\right\}$ denote the Markov sequence describing the random ancestral relations of n gene copies (leaves) sampled from a population of size N evolving according to Def 3.1. If the random number of potential offspring (X) of an arbitrary individual is distributed according to $(C, \alpha > 0$ fixed)

$$\lim_{x\to\infty} Cx^{\alpha}\mathbb{P}\left(X \geq x\right) = 1 \tag{1}$$

[Sch03, Eq 11] then $\left\{\xi^{N,n}(\lfloor t/c_N \rfloor); t \geq 0\right\}$ converges as $N \to \infty$ in the Skorokhod topology to the Beta$(2-\alpha, \alpha)$-coalescent provided $1 \leq \alpha < 2$ [Sch03, Thm 4c].

Suppose X is distributed according to ($[n] \equiv \{1, 2, \ldots, n\}$ for $n \in \mathbb{N} \equiv \{1, 2, \ldots\}$)

$$g(k)\left(\frac{1}{k^a} - \frac{1}{(1+k)^a}\right) \leq \mathbb{P}\left(X = k\right) \leq f(k)\left(\frac{1}{k^a} - \frac{1}{(1+k)^a}\right), \quad k \in [\zeta(N)], \tag{2}$$

where the functions g, f are independent of N and so that $\mathbb{E}\left[X\right] > 1$ and $\mathbb{P}\left(X \leq \zeta(N)\right) = 1$. Write $X \rhd \mathbb{L}(a, \zeta(N))$ when X is distributed according to (2) with a and $\zeta(N)$ as given each time. Furthermore, the population evolves according to Definition 3.2 or Definition 3.3.

Definition 3.2 (The random environment of type A) *Suppose a population evolves according to Def. 3.1 with the number of potential offspring produced by each individual distributed according to* (1) *or* (2)*. Fix $0 < \alpha < 2 \leq \kappa$. Write E for the event $\{X_i \rhd \mathbb{L}(\alpha, \zeta(N))$ for all $i \in [N]\}$ and E^{c} when κ replaces α in E for some given $\zeta(N)$. Suppose, with $(\varepsilon_N)_N$ a positive sequence with $0 < \varepsilon_N < 1$ and $\varepsilon_N \to 0$,*

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

Definition 3.3 (The random environment of type B) *Suppose a population evolves according to Definition 3.1 with the law for the number of potential offspring given by* (2)*. Fix $0 < \alpha < 2 \leq \kappa$. Write E_1 for the event*

$$\{\textit{there exists exactly one } i \in [N] \textit{ where } X_i \rhd \mathbb{L}(\alpha, \zeta(N)), \textit{ and } X_j \rhd \mathbb{L}(\kappa, \zeta(N)) \textit{ all } j \in [N] \setminus \{i\}\}$$

with i picked uniformly at random, and E_1^{c} when κ replaces α in E_1 so that $E_1^{\mathsf{c}} = E^{\mathsf{c}}$ from Definition 3.2. Suppose

$$\mathbb{P}\left(E_1\right) = \overline{\varepsilon}_N, \quad \mathbb{P}\left(E_1^{\mathsf{c}}\right) = 1 - \overline{\varepsilon}_N$$

where $(\overline{\varepsilon}_N)_N$ is a positive sequence with $0 < \overline{\varepsilon}_N < 1$ for all N and it may hold that $\overline{\varepsilon}_N \to 0$ as $N \to \infty$.

Then $\{\xi^{N,n}(\lfloor t/c_N \rfloor); t \geq 0\}$ converges as $N \to \infty$ in the Skorohod topology to the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent provided $\liminf_{N\to\infty} \zeta(N)/N > 0$. One may identify conditions on ε_N and $\overline{\varepsilon}_N$ such that the models described in Definition 3.1 and Definition 3.3 and Definition 3.2 are in the domain of attraction of the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent [Eld24].

Let $\{\xi^n(t) : t \geq 0\}$ be a given coalescent, a Markov chain tracking the random relations of sampled gene copies, $\#A$ the cardinality of a given set A, $L_i^N(n) \equiv \int_0^{\tau(n)} \#\{\xi \in \xi^n(t) : \#\xi = i\}\, dt$ and $\tau(n) \equiv \inf\{t \geq 0 : \#\xi^n(t) = 1\}$ for $i \in \{1, 2, \ldots, n-1\}$; $R_i(n) \equiv L_i(n)/\sum_j L_j(n)$ for $i = 1, 2, \ldots, n-1$. Then $L_i^N(n)$ is interpreted as the random total length of branches supporting $i \in \{1, 2, \ldots, n\}$ leaves, with the length measured in generations, and n sample size. We then have $L(n) = L_1(n) + \cdots + L_{n-1}(n)$.

We estimate

$$\varrho_i(n) := \mathbb{E}\left[\frac{L_i(n)}{L(n)}\right], \quad i = 1, 2, \ldots, n-1 \tag{3}$$

for the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent with transition rates

$$\lambda_{n,k} = \mathbb{1}_{\{k=2\}}\binom{n}{k}\frac{C_\kappa}{C_{\alpha,\gamma}} + \binom{n}{k}\frac{\alpha c'_\alpha}{C_{\alpha,\gamma} m_\infty^\alpha} B(\gamma, k-\alpha, n-k+\alpha) \tag{4a}$$

$$\gamma = \mathbb{1}_{\left\{\frac{\zeta(N)}{N} \to K\right\}}\frac{K}{m_\infty + K} + \mathbb{1}_{\left\{\frac{\zeta(N)}{N} \to \infty\right\}} \tag{4b}$$

$$C_{\alpha,\gamma} = C_\kappa + \frac{\alpha c}{m_\infty^\alpha}\int_{\mathbb{1}_{\{\zeta(N)/N \to K\}}\frac{m_\infty}{K+m_\infty}}^1 (1-y)^{1-\alpha} y^{\alpha-1} dy = C_\kappa + \frac{\alpha c}{m_\infty^\alpha} B(\gamma, 2-\alpha, \alpha) \tag{4c}$$

$$C_\kappa = \mathbb{1}_{\{\kappa=2\}} m_\infty^{-2} + \mathbb{1}_{\{\kappa>2\}} m_\infty^{-2} c_\kappa \tag{4d}$$

The range of α in Def 3.2 and Def 3.3 intersects at 1 so one must decide which of the two definitions holds when $\alpha = 1$. Let where $0 < c' < 1$ and $c > 0$. Write $c'_\alpha = c$ if Def 3.2 holds when $\alpha = 1$, and $c'_\alpha = \mathbb{1}_{\{\alpha=1\}} c' + \mathbb{1}_{\{\alpha \in (0,2)\setminus\{1\}\}} c$ when Def 3.3 holds when $\alpha = 1$ (and $\kappa > 2$). We approximate m_∞ with $m_\infty \approx (2 + (1 + 2^{1-\kappa})/(\kappa - 1))/2$. For $\kappa > 2$ we can approximate $c_\kappa \approx (c'_\kappa + c''_\kappa)/2$ where

$$c'_\kappa = \frac{\kappa 2^{2-\kappa}}{\kappa - 2} - \frac{3\kappa 2^{1-\kappa}}{\kappa - 1} + 2^{-\kappa}, \quad c''_\kappa = \frac{\kappa 2^{2-\kappa}}{\kappa - 2} - \frac{\kappa 2^{1-\kappa}}{\kappa - 1}$$

When $\zeta(N)/N \to 0$ we have, as one would expect, $\gamma = 0$ and so $B(\gamma, 2-\alpha, \alpha) = 0$ and $C_{\alpha,\gamma} = C_\kappa$ and thus $\lambda_{n,k} = \mathbb{1}_{\{\kappa=2\}}\binom{n}{k}$, the transition rate of the Kingman-coalescent.

In § 4 we summarize the algorithm. Our goal is simply to arrive at a working code that is correct. The code is partitioned into the functions described in § 4.1–§ 4.14; we conclude in § 5. Comments within the code are shown in **this font and colour.**

4 code

We use the GSL library. It suffices to keep track of the current block sizes $(b_1, \ldots, b_m)$ where $b_j \geq 1$ and $\sum_j b_j = n$. Let n denote the sample size and m the current number of blocks. Let $\ell_i(n)$ for $i = 1, 2, \ldots, n-1$ denote realised branch lengths.

1. $(r_1(n), \ldots, r_{n-1}(n)) \leftarrow (0, \ldots, 0)$
2. for each of M experiments § 4.13 :
   (a) initialize block sizes $(b_1, \ldots, b_n) \leftarrow (1, \ldots, 1)$
   (b) initialise branch lengths $(\ell_i(n), \ldots, \ell_{n-1}(n)) \leftarrow (0, \ldots, 0)$
   (c) set the current number of blocks to sample size $m \leftarrow n$
   (d) **while** $m > 1$:
      i. sample exponential time t with rate (4a) $\lambda_m = \lambda_{m,2} + \cdots + \lambda_{m,m}$ § 4.12
      ii. update branch lengths $\ell_{b_j}(n) \leftarrow t + \ell_{b_j}(n)$ for $j = 1, 2, \ldots, m$ § 4.11
      iii. sample merger size $k = \min\{j : u \leq \sum_{i=2}^{j} \lambda_{i,m}/\lambda_m\}$ § 4.9
      iv. merge blocks and record size $b_{\sigma(1)} + \cdots + b_{\sigma(k)}$ of continuing block § 4.12
      v. given merger size update current number of blocks $m \leftarrow m - k + 1$
   (e) given a realisation of branch lengths update estimate of $\varrho_i(n)$ § 4.10 :

$$r_i(n) \leftarrow r_i(n) + \frac{\ell_i(n)}{\sum_{j=1}^{n-1} \ell_j(n)}$$

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

4.1 the included libraries

the included C++ libraries

5 ⟨ includes 5 ⟩ ≡

```
#include <iostream>
#include <vector>
#include <random>
#include <functional>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <list>
#include <forward_list>
#include <cassert>
#include <math.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_sf.h>
#include <gsl/gsl_cdf.h>
```

This code is used in chunk 18.

4.2 GSL random number generator

6 ⟨ rngs 6 ⟩ ≡

```
std::random_device randomseed;      /*
      Standard mersenne twister random number engine seeded with rng() */
std::mt19937_64 rng(randomseed());      /*
      define a GSL random number generator */
gsl_rng * rngtype;
static void setup_rng(const unsigned long 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 18.

4.3 the beta function

return the logarithm of the (incomplete) beta function $B(x,a,b) = \int_0^x t^{a-1}(1-t)^{b-1}dt$ for $0 < x \leq 1$ and $a,b > 0$ using the Gauss hypergeometric function

$$B(x,a,b) = \frac{1}{a}x^a(1-x)^b F(a+b,1,a+1,x)$$

7 ⟨ beta function 7 ⟩ ≡

```
static long double betafunc(const double &x, const double &a, const double &b)
{   /*
      the GSL incomplete beta function is normalised by the complete beta function */
  assert(x > 0);
  assert(a > 0);
  assert(b > 0);   /*
      the standard way would be gsl_sf_beta_inc(a, b, x) * gsl_sf_beta(a, b) */   /*
      return the logarithm of the beta function as log Γ(a) + log Γ(b) − log Γ(a + b) */
  const long double f = static_cast⟨long double⟩((x < 1 ? gsl_sf_hyperg_2F1(a + b, 1,
      a + 1, x) : 1));
  assert(f > 0);   /*
      return 𝟙{x<1}(log f + a log x + b log(1 − x) − log a) + 𝟙{x=1}(log Γ(a) + log Γ(b) − log Γ(a + b))
      */
  return (x < 1 ? (logl(f)+(static_cast⟨long double⟩((a*log(x))+(b*log(1−x))−log(a)))) :
      (lgammal(static_cast⟨long double⟩(a)) + lgammal(static_cast⟨long
      double⟩(b)) − lgammal(static_cast⟨long double⟩(a + b))));
}
```

This code is used in chunk 18.

4.4 binomial constant

compute the log of the binomial constant $\binom{n}{k}$

8 ⟨ binomial 8 ⟩ ≡

```
static long double binom(const int &n, const int &k)
{
  assert(k ≤ n);
  assert(k ≥ 0);
  assert(n ≥ 0);      /*
      log (n k) = log Γ(n + 1) − log Γ(k + 1) − log Γ(n − k + 1)  */
  return (lgammal(static_cast⟨long double⟩(n + 1)) − lgammal(static_cast⟨long
      double⟩(k + 1)) − lgammal(static_cast⟨long double⟩(n − k + 1)));
}
```

This code is used in chunk 18.

4.5 total beta rate

return the total beta rate $\binom{n}{k} B(\gamma, k-\alpha, n+\alpha-k)$.

9 ⟨ betarate 9 ⟩ ≡

```
static double lbetank(const int &n, const int &k, const double &a, const double &g)
{    /*
       return exp(log (n k) + log B(γ, k − α, n − k + α)) */    /*
       binom § 4.4 and betafunc § 4.3 */
  return (static_cast⟨double⟩(expl(binom(n, k) + betafunc(g, static_cast⟨double⟩(k) − a,
      static_cast⟨double⟩(n − k) + a))));
}
```

This code is used in chunk 18.

4.6 the constant $C_{\alpha,\gamma}$ (4c)

compute the constant $C_{\alpha,\gamma}$ (4c) where we approximate $m_\infty \approx (2+(1+2^{1-\kappa})/(\kappa-1))/2$

10 ⟨ constantC 10 ⟩ ≡

```
static double constantC(const double &a, const double &kappa, const double &c, const
        double &cprime, const double &g)
{   /*
      a = α, k = κ, c = c, cprime = c′ */   /*
      g = γ = 𝟙_{ζ(N)/N→K} K/(m_∞+K) + 𝟙_{ζ(N)/N→∞} and m_∞/(K+m_∞) = 1 − γ */
  const double m = (2. + (1. + pow(2, 1 − kappa))/(kappa − 1))/2.;   /*
      (4d); first taking κ = 2 */
  const double Ck = 1/pow(m, 2.);   /*
      c′_α = 𝟙_{α=1} c′ + 𝟙_{α∈(0,2)\{1}} c */
  const double caprime = (a < 1 ? c : (a > 1 ? c : cprime));   /*
      the integral in (4c) B(γ, 2 − α, α) = ∫_0^{𝟙_{ζ(N)/N→K} m_∞/(K+m_∞)} (1 − y)^{1−α} y^{α−1} dy */   /*
      the incomplete beta function is normalised by the complete beta function in GSL
     */
  const double tegur = (g < 1 ? (gsl_sf_beta_inc(2 − a, a, g) ∗ gsl_sf_beta(2 − a,
      a)) : gsl_sf_beta(2 − a, a));
  return (Ck + (a ∗ caprime ∗ tegur/pow(m, a)));
}
```

This code is used in chunk 18.

4.7 $\lambda_{n,k}$ (4a)

compute the transition rate $\lambda_{n,k}$ in (4a)

11 ⟨ lambdank 11 ⟩ ≡

```
static double lambdank(const int &n, const int &k, const double &a, const double
        &kappa, const double &c, const double &cprime, const double &gm)
{
  assert(k > 1);
  assert(n > 1);
  assert(k ≤ n);      /*
      we approximate m_∞ ≈ (1/2)(2 + (1+2^{1−κ})/(κ−1)) */
  const double m = (2. + (1. + pow(2., 1. − kappa))/(kappa − 1.))/2.;      /*
      Ck is C_κ (4d); first taking κ = 2 */
  const double Ck = 2/pow(m, 2.);
  const double caprime = (a > 1 ? c : (a < 1 ? c : cprime));      /*
      return (4a) using lbetank § 4.5 and constantC § 4.6 */
  return (((k < 3 ? (Ck * (static_cast⟨double⟩(k * (k − 1)))/2.) : 0) + (a * caprime * lbetank(n,
      k, a, gm)/pow(m, a)))/constantC(a, kappa, c, cprime, gm));
}
```

This code is used in chunk 18.

4.8 the total rate λ_n

compute the total jump rate $\lambda_n = \sum_k \lambda_{n,k}$ with $\lambda_{n,k}$ as in (4a)

12 ⟨ jump rate λ_n 12 ⟩ ≡

```
static void lambdan (const int &n, const double &ia, const double &ikappa, const
    double &ic, const double &icpr, const double &ig, std::vector < double > &l )
{
  assert(n > 1);
  for (int i = 2; i ≤ n; ++i) {      /*
        compute λ_i for i = 2, 3, . . . , n */
    l[i] = 0;
    for (int j = 2; j ≤ i; ++j) {      /*
          compute λ_i */      /*
          lambdank § 4.7 */
      l[i] += lambdank(i, j, ia, ikappa, ic, icpr, ig);
    }
    assert(l[i] > 0);
  }
}
```

This code is used in chunk 18.

4.9 sample merger size

sample merger size $\min\{j \in \{2,3,\ldots,m\} : \sum_{k=2}^{j} \lambda_{n,k} / \sum_{i=2}^{m} \lambda_i \geq U\}$ for a given random uniform U

13 ⟨ merger size 13 ⟩ ≡

```
static int samplemerger (const int &m, const double &ia, const double &ikappa, const
    double &ic, const double &icpr, const double &ig, const std::vector < double > &l
    )
{   /*
      m is current number of blocks */
  assert(m > 1);
  int j = 2;
  const double u = gsl_rng_uniform(rngtype);   /*
      lambdank § 4.7 */
  double s = lambdank(m, 2, ia, ikappa, ic, icpr, ig);
  assert(l[m] > 0);
  while (u > (s/l[m])) {
    ++j;
    s += lambdank(m, j, ia, ikappa, ic, icpr, ig);
  }   /*
        return the size of the merger */
  return j;
}
```

This code is used in chunk 18.

4.10 update estimate of $\varrho_i(n)$ (3)

update estimate of $\varrho_i(n)$; i.e. given a realisation $r(n)$ of $(R_1(n), \ldots, R_{n-1}(n))$ update the estimate $e_i(n) \leftarrow e_i(n) + r_i(n)$ where $r_i(n) = \ell_i(n) / \sum_j \ell_j(n)$

14 ⟨ update $\varrho_i(n)$ 14 ⟩ ≡

```
static void updateri (const int &leaves, const std::vector < double > &tLi, std::vector <
    double > &eri )
{    /*
       tLi is the the current realisation ℓ_i(n) of L_i(n)  */    /*
       tLi[0] is the current total tree length Σ_i ℓ_i(n)  */
  assert(tLi[0] > 0);
  for (int i = 1; i < leaves; ++i) {    /*
         for checking return estimates of 𝔼[L_i(n)]/𝔼[L(n)]; tLi[0] records estimates of
       𝔼[L(n)]  */    /*
         update with tLi[i]/tLi[0] when comparing with 𝔼[R_i^N(n)]  */
    eri[i] += (tLi[i]/tLi[0]);
  }
}
```

This code is used in chunk 18.

4.11 update current branch length

update current branch lengths $\ell_i(n)$

15 ⟨ update current lengths 15 ⟩ ≡

```
static void updatells (const double &qi, const std::vector < int > &tree, std::vector <
    double > &elli ) {      /*
     qi is the total jump rate λ_m = Σ_{j=2}^{m} λ_{j,m} (4a) */
assert(qi > 0);      /*
     sample sojourn time in current state */
const double timi = gsl_ran_exponential(rngtype, 1./qi);
assert(timi > 0); for (const auto &b:tree)
{      /*
        each block size b must be at least 1 */
  assert(b > 0);
  elli[0] += timi;
  elli[b] += timi;
}
assert(elli[0] > 0); }
```

This code is used in chunk 18.

4.12 gene genealogy and branch lengths

generate one gene genealogy or tree and record the branch lengths of the tree

16 ⟨ genealogy 16 ⟩ ≡

```
static void genealogy (const int &leaves, const double &ia, const double &ikappa, const
    double &ic, const double &icpr, const double &ig, const std::vector <
    double > &lambdab, std::vector < double > &eri ) {      /*
     lambdab is λ_n the total jump rate */
std::vector < int > tree(leaves, 1); std::vector < double > tli(leaves + 1, 0);
double timi
{ }
;
int mergersize
{ }
;
int sizenewblock
{ }
;
int b
{ }
;
assert(tree.size( ) ≡ leaves);
while (tree.size( ) > 1) {      /*
        update the current branch lengths ℓ_i(n) § 4.11 */
  updatells(lambdab[tree.size( )], tree, tli);      /*
        sample merger size number of blocks merging § 4.9 */
  mergersize = samplemerger(static_cast⟨int⟩(tree.size( )), ia, ikappa, ic, icpr, ig, lambdab);
  assert(mergersize > 1);
  std::shuffle(std::begin(tree), std::end(tree), rng);
  sizenewblock = static_cast⟨int⟩(std::accumulate(tree.rbegin( ), std::next(tree.rbegin( ),
      mergersize), 0));
  assert(sizenewblock > 1);
  b = static_cast⟨int⟩(tree.size( ));      /*
        remove the merged blocks from the tree */
  assert(mergersize ≤ b);
  tree.resize(b − mergersize);      /*
        add the new block to the tree */
  tree.push_back(sizenewblock);
}
assert(tree.size( ) < 2);      /*
      generated one realisation of branch lengths; update estimate of ϱ_i(n) § 4.10 */
updateri(leaves, tli, eri); }
```

This code is used in chunk 18.

4.13 estimate $\varrho_i(n)$

estimate $\varrho_i(n)$

17 ⟨ estimate 17 ⟩ ≡

```
static void estimate(const int &n, const int &nexperiments, const double &ia, const
        double &ikappa, const double &ic, const double &icpr, const double &ig) {
  assert(n > 1);      /*
      initialise the vector for the total jump rate λ_i */
  std::vector < double > il(n + 1, 0);      /*
      record the jump rates lambdan § 4.8 */
  lambdan(n, ia, ikappa, ic, icpr, ig, il);      /*
      veri records the estimates of ϱ_i(n) */
  std::vector < double > veri(n);
  int r = nexperiments + 1;
  while (--r > 0) {      /*
      § 4.12 */
    genealogy(n, ia, ikappa, ic, icpr, ig, il, veri);
  }      /*
      print the estimate of ϱ_i(n) */
  for (const auto &x:veri)
  {
    std::cout ≪ x ≪ '\n';
  }
}
```

This code is used in chunk 18.

4.14 the main function

```
18      /*
        § 4.1 */
    ⟨ includes 5 ⟩  /*
        § 4.2 */
    ⟨ rngs 6 ⟩  /*
        § 4.3 */
    ⟨ beta function 7 ⟩  /*
        § 4.4 */
    ⟨ binomial 8 ⟩  /*
        § 4.5 */
    ⟨ betarate 9 ⟩  /*
        § 4.6 */
    ⟨ constantC 10 ⟩  /*
        § 4.7 */
    ⟨ lambdank 11 ⟩  /*
        § 4.8 */
    ⟨ jump rate λ_n 12 ⟩  /*
        § 4.9 */
    ⟨ merger size 13 ⟩  /*
        § 4.10 */
    ⟨ update ϱ_i(n) 14 ⟩  /*
        § 4.11 */
    ⟨ update current lengths 15 ⟩  /*
        § 4.12 */
    ⟨ genealogy 16 ⟩  /*
        § 4.13 */
    ⟨ estimate 17 ⟩
    int main(int argc, char *argv[])
    {  /*
          setup_rng § 4.2 */
      setup_rng(static_cast⟨unsigned long⟩(atoi(argv[1])));  /*
          set the parameter values */  /*
          set α */
      const double setalpha = 0.01;  /*
          set κ */
      const double setkappa = 2;
      const double setc = 1;
      const double setcprime = 1;  /*
          set γ */
      const double setgamma = 1.0;  /*
          estimate from § 4.13 */
      estimate(100, 200000, setalpha, setkappa, setc, setcprime, setgamma);
      gsl_rng_free(rngtype);
      return GSL_SUCCESS;
    }
```

5 conclusions and bibliography

We estimate the functionals $\mathbb{E}[R_i(n)]$ when the coalescent $\{\xi^n; t \geq 0\}$ is the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent. The δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent can be obtained from a population model of a haploid panmictic population of constant size evolving according to sweepstakes reproduction (heavy right-tailed offspring number distribution) in a random environment (Definitions 3.2 and 3.3). The estimates of $\mathbb{E}[R_i(n)]$ can be compared to estimates of $\mathbb{E}\left[R_i^N(n)\right]$, funcionals of an ancestral process in the domain-of-attraction of $\{\xi^n\}$.

Figure 1 records an example of an approximation of $\mathbb{E}[R_i(n)]$

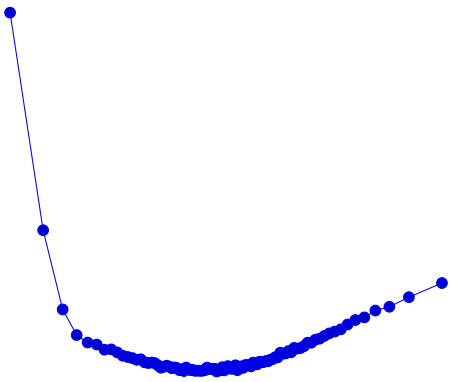

Figure 1: *An example approximation of* $\mathbb{E}[R_i(n)]$ *graphed as logits against* $\log(i/n) - log(1-i/n)$ *where* n *is sample size*

References

[Eld24] Bjarki Eldon. Gene genealogies in haploid populations evolving according to sweepstakes reproduction. In preparation, 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.

[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

⟨ beta function 7 ⟩ Used in chunk 18.
⟨ betarate 9 ⟩ Used in chunk 18.
⟨ binomial 8 ⟩ Used in chunk 18.
⟨ constantC 10 ⟩ Used in chunk 18.
⟨ estimate 17 ⟩ Used in chunk 18.
⟨ genealogy 16 ⟩ Used in chunk 18.
⟨ includes 5 ⟩ Used in chunk 18.
⟨ jump rate λ_n 12 ⟩ Used in chunk 18.
⟨ lambdank 11 ⟩ Used in chunk 18.
⟨ merger size 13 ⟩ Used in chunk 18.
⟨ rngs 6 ⟩ Used in chunk 18.
⟨ update $\varrho_i(n)$ 14 ⟩ Used in chunk 18.
⟨ update current lengths 15 ⟩ Used in chunk 18.