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

Bjarki Eldon[1][2]

Let $\{\xi^n\} \equiv \{\xi^n(t); t \geq 0\}$ denote the Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent for $0 < \alpha < 2$ and $0 < \gamma \leq 1$. Write $L_i(n) \equiv \int_0^{\tau(n)} \#\{\xi \in \xi^n(t) : \#\xi = i\}\, dt$, $L(n) \equiv \int_0^{\tau(n)} \#\xi^n(t) dt$, $\tau(n) \equiv \inf\{t \geq 0 : \#\xi^n(t) = 1\}$, and $R_i(n) \equiv L_i(n)/L(n)$ for $i = 1, 2, \ldots, n-1$. With this C++ code we estimate the functionals $\mathbb{E}[R_i(n)]$ for $i = 1, 2, \ldots, 2n-1$ when the gene genealogy of $2n$ sampled gene copies is described by the Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent for $0 < \alpha < 2$ and $0 < \gamma \leq 1$. The stated coalescent can be obtained from a population genetics model of a diploid panmictic population of constant size evolving absent selfing and according to sweepstakes reproduction, and extends the Ω-Beta$(2-\alpha, \alpha)$-coalescent [BLS15]. The Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent is a continuos-time simultaneous multiple-merger coalescent with an atom at zero and where the ancestral lineages (blocks of a partition of $\{1, 2, \ldots, 2n\}$) merging in a single group are independently and uniformly at random and with replacement assigned a label from $\{1, 2, 3, 4\}$ and the blocks assigned the same label are merged. Only one such group of blocks can appear each time, corresponding to the appearance of one large family involving four parent chromosomes.

Contents

[1] beldon11@gmail.com

[2] Supported by Deutsche Forschungsgemeinschaft (DFG) - Projektnummer 273887127 through DFG SPP 1819: Rapid Evolutionary Adaptation grant STE 325/17 to Wolfgang Stephan; acknowledge funding by the Icelandic Centre of Research (Rannís) through an Icelandic Research Fund (Rannsóknasjóður) Grant of Excellence no. 185151-051 to Einar Árnason, Katrín Halldórsdóttir, Alison M. Etheridge, Wolfgang Stephan, and BE; 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 [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 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
     c++ -Wall -Wextra -pedantic -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 --leak-resolution=high
--num-callers=40 --vgdb=full <program start>
```

Use `cppcheck` to check the code

```
cppcheck --enable=all --language=c++ iguana.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

Write $(a)_n := a(a-1)\cdots(a-n+1)$ with $(a)_0 := 1$, and for any given event/condition E let $\mathbb{1}_{\{E\}} := 1$ whenever E occurs/holds, and take $\mathbb{1}_{\{E\}} = 0$ otherwise; write $[n] := \{1, 2, \ldots, n\}$ for any $n \in \mathbb{N} := \{1, 2, \ldots\}$. Consider a Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent on the partitions of $[2n]$ for $n \in \{2, 3, \ldots\}$; we interpret n as the number of diploid individuals and so $2n$ gene copies sampled. Write $s = m - k_1 - \ldots - k_4$ for $k_1, \ldots, k_4 \in \{0, 2, 3, \ldots, m\}$ and $2 \leq k_1 + \cdots + k_4 \leq m$ for any $m \in \{2, 3, \ldots, n\}$. The Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent has transition rates

$$\lambda_{m;k_1,\ldots,k_4;s} = \binom{m}{k_1 \ldots k_4 \quad s} \frac{1}{\prod_{j=2}^{m} \left(\sum_i \mathbb{1}_{\{k_i=j\}}\right)!} \lambda'_{m,k_1,\ldots,k_4,s} \tag{1}$$

where $\lambda_{m;k_1,\ldots,k_4;s}$ is the total rate of merging blocks in groups of sizes $k_1, \ldots, k_4$ and $r = \sum_j \mathbb{1}_{\{k_j \geq 2\}}$

$$\lambda'_{m,k_1,\ldots,k_4,s} = \mathbb{1}_{\left\{\sum_j k_j = 2\right\}} \frac{C_\kappa}{C_{\alpha,\gamma}} + \frac{\alpha c}{C_{\alpha,\gamma} \mathrm{m}^\alpha} \sum_{\ell=0}^{s \wedge (4-r)} \binom{s}{\ell} \frac{(4)_{r+\ell}}{4^{k+\ell}} B(\gamma, k+\ell-\alpha, m-k-\ell+\alpha) \tag{2}$$

where $B(p, a, b) = \int_0^1 \mathbb{1}_{\{0<u\leq p\}} u^{a-1}(1-u)^{b-1} du$ for $0 < p \leq 1$ and $a, b > 0$ and

$$\mathrm{m} = (2 + (1 + 2^{1-\kappa})/(\kappa - 1))/2 \tag{3a}$$

$$\gamma = \mathbb{1}_{\left\{\frac{\zeta(N)}{N} \to K\right\}} \frac{K}{K + \mathrm{m}} + \mathbb{1}_{\left\{\frac{\zeta(N)}{N} \to \infty\right\}} \tag{3b}$$

$$C_{\alpha,\gamma} = \frac{1}{4} C_\kappa + \frac{\alpha c}{4 \mathrm{m}^\alpha} B(\gamma, 2-\alpha, \alpha) \tag{3c}$$

$$C_\kappa = \mathbb{1}_{\{\kappa=2\}} \frac{2}{\mathrm{m}^2} + \mathbb{1}_{\{\kappa>2\}} \frac{2}{\mathrm{m}^2} \frac{c_\kappa}{2^\kappa (\kappa-2)(\kappa-1)} \tag{3d}$$

where in (3d) $\kappa + 2 < c_\kappa < \kappa^2$ when $\kappa > 2$. In (2) $c > 0$ and $0 < \alpha < 2$.

In (2) we take $0 < \alpha < 2$ with the understanding that the population model when $0 < \alpha < 1$ is different from the one when $1 \leq \alpha < 2$. Let X be a positive integer-valued random variable with law

$$\mathbb{P}(X = k) = C\left(k^{-\alpha} - (1+k)^{-\alpha}\right) \tag{4}$$

for $k \in \{2, 3, \ldots, \zeta(N)\}$ where C is such that $\mathbb{P}(2 \leq X \leq \zeta(N)) = 1$; X is the random number of potential offspring of an arbitrary parent pair. In any given generation it is assumed that the $2N$ current individuals randomly form parent pairs, and the pairs then independently produce potential offspring according to (4). Out of the at least $2N$ potential offspring so generated we sample $2N$ uniformly at random and without replacement to survive and replace the current individuals. Write $X \rhd \mathbb{L}(a, \zeta(N))$ when X is given law (4) with a and $\zeta(N)$ as given each time. Write $X_1(g), \ldots, X_N(g)$ for the random number of potential offspring produced in generation g, and $(U_g)_g$ for a sequence of i.i.d. random uniforms and $(\varepsilon_N)_N$ where $0 < \varepsilon_N < 1$. Suppose $\kappa, \zeta(N) \geq 2$ fixed and $X_1(g), \ldots, X_N(g)$ are iid copies of $X(g)$ where

$$X(g) \rhd \mathbb{L}\left(\mathbb{1}_{\{U_g \leq \varepsilon_N\}} \alpha + \mathbb{1}_{\{U_g > \varepsilon_N\}} \kappa, \zeta(N)\right) \tag{5}$$

when $1 \leq \alpha < 2$; when $0 < \alpha < 1$ it holds that

$$X_i(g) \rhd \mathbb{L}\left(\mathbb{1}_{\{U_g \leq \varepsilon_N\}} \alpha + \mathbb{1}_{\{U_g > \varepsilon_N\}} \kappa, \zeta(N)\right), \quad X_{j \neq i}(g) \rhd \mathbb{L}(\kappa, \zeta(N)) \tag{6}$$

In (6) the index i is picked uniformly at random from $[N]$, and $X_j(g) \rhd L(\kappa, \zeta(N))$ for all $j \neq i$. Both (5) and (6) lead to (2), so that when we use (2) with $0 < \alpha < 2$ it is with the understanding that (6) holds when $0 < \alpha < 1$, and when $1 \leq \alpha < 2$ (5) is in force. In (5) the $X_1, \ldots, X_N$ are i.i.d.; they are independent but may not always be identically distributed when (6) holds.

Let $\{\xi^n\} \equiv \{\xi^n(t); t \geq 0\}$ be the coalescent on the partitions of $[2n]$ with transitions rates (2) and $\xi^n(0) = \{\{1\}, \ldots, \{2n\}\}$. Define the functionals, where $\#A$ is the cardinality of a given set A,

$$L_i(n) := \int_0^{\tau(n)} \#\{\xi \in \xi^n(t) : \#\xi = i\}\, dt, \quad L(n) := \int_0^{\tau(n)} \#\xi^n(t) dt \tag{7}$$

where $i \in [2n-1]$ and $\tau(n) := \inf\{t \geq 0 : \#\xi^n(t) = 1\}$. Then $L_i(n)$ are the random length of branches supporting i leaves (gene copies), and $L(n) = L_1(n) + \cdots + L_{2n-1}(n)$ is the random total tree length[BLS15]. Define, for $n \in \{2, 3, \ldots\}$,

$$R_i(n) := \frac{L_i(n)}{L(n)}, \quad i = 1, 2, \ldots, 2n-1 \tag{8}$$

and $R_i(n)$ is well defined since $L(n) > 0$ almost surely. We will estimate $\mathbb{E}[R_i(n)]$ when $\{\xi^n\}$ has transition rates (1) with $\lambda_{n;k_1,\ldots,k_r;s}$ as in (2).

In § 4 we briefly summarize the algorithm; the code follows in § 4.1–§ 4.18; we conclude in § 5.

4 code

Let

$$\mathbb{K}_m := \{(k_1, \cdots, k_4) : k_j \in \{0, 2, 3, \ldots, m\}, \quad k_1 \geq k_2 \geq k_3 \geq k_4, \quad 2 \leq k_1 + \cdots + k_4 \leq m\}$$

be the set of all possible ordered merger sizes when there are m blocks and $j_m : \mathbb{K}_m \to [\#\mathbb{K}_m]$ be a bijection assigning indexes to the mergers ordered such that (see § 4.10 for the ordering)

$$1 = j_m((2,0,0,0)) < j_m((2,2,0,0)) < \ldots < j_m((m,0,0,0)) = \#\mathbb{K}_m.$$

Let $s = m - \sum_j k_j$ and $\lambda_m := \sum_{(k_1,\ldots,k_4)\in\mathbb{K}_m} \lambda_{m;k_1,\ldots,k_4;s}$ denote the total jump rate out of m blocks (recall (1)). Let $F_m : [\#\mathbb{K}_m] \to [0,1]$ denote the cumulative mass function

$$F_m(\ell) = \frac{1}{\lambda_m} \sum_{i=1}^{\ell} \lambda_{m;j_m^{-1}(i);s}$$

We use F_m to sample merger sizes. Let U denote a standard random uniform. Then, given a sample of U, $j_m^{-1}(j^*) \in \mathbb{K}_m$ where $j^* := \inf\{j \in [\#\mathbb{K}_m] : U \leq F_m(j)\}$.

Let $(\ell_i(n), \ldots, \ell_{n-1}(n))$ denote the realised branch lengths (7), and $(b_1, \ldots, b_m)$ the current block sizes where $b_j \in [2n]$ and $\sum_j b_j = 2n$ (the sample consists of n diploid individuals and so $2n$ gene copies).

1. $(\mathfrak{r}_1(n), \ldots, \mathfrak{r}_{2n-1}(n)) \leftarrow (0, \ldots, 0)$
2. compute λ_m for $m = 2, 3, \ldots, 2n$ given parameter values in § 4.2; § 4.11
3. for each of M experiments § 4.17
   (a) $(\ell_1(n), \ldots, \ell_{2n-1}(n)) \leftarrow (0, \ldots, 0)$
   (b) $m \leftarrow 2n$ where m is current number of blocks and $2n$ is sample size
   (c) $(b_1, \ldots, b_{2n}) \leftarrow (1, \ldots, 1)$
   (d) **while** $m > 1$: § 4.16
      i. sample a random exponential t with rate λ_m
      ii. $\ell_b(n) \leftarrow \ell_b(n) + t$ for $b = b_1, \ldots, b_m$ § 4.14
      iii. sample merger sizes $j_m^{-1}(j^*) \in \mathbb{K}_m$ where $j^* := \inf\{j \in [\#\mathbb{K}_m] : U \leq F_m(j)\}$ § 4.12
      iv. merge blocks according to sampled merger sizes $j_m^{-1}(j^*) \in \mathbb{K}_m$ § 4.13
      v. $m \leftarrow m - \sum_i k_i + \sum_j \mathbb{1}_{\{k_j \geq 2\}}$ where $(k_1, \ldots, k_4) = j_m^{-1}(j^*)$
   (e) $\mathfrak{r}_i(n) \leftarrow \mathfrak{r}_i(n) + \ell_i(n) / \sum_j \ell_j(n)$ for $i \in [2n-1]$ § 4.15
4. return an estimate $\overline{\varrho}_i(n) = (1/M)\mathfrak{r}_i(n)$ of $\mathbb{E}[R_i(n)]$

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 <limits>
#include <cfloat>
#include <assert.h>
#include <math.h>
#include <fenv.h>
#include <errno.h>
#include <unistd.h>
#include <omp.h>
#include <sys/param.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_sf.h>
#include "xindbetacoal.h"
```

This code is used in chunk 22.

4.2 constants

the parameter values

6 ⟨ constants 6 ⟩ ≡

```
const double dblepsilon = 2.2204460492503131 · 10^-16;      /*
    0 < α < 2 */
const double CONST_ALPHA = 0.5;      /*
    γ (3b) */
const double CONST_GAMMA = 0.5;
const double CONST_C = 1.;      /*
    κ ≥ 2 */
const double c_kappa = 2.;      /*
    m = (2 + (1 + 2^(1−κ))/(κ − 1))/2 (3a) */
const double CONST_Minf = (2. + (1 + pow(2., 1. − c_kappa))/(c_kappa − 1.))/2.;      /*
    (κ + 2 + κ^2)/(2^(1+κ)(κ − 2)(κ − 1)) = c_κ */
const double ck = (2. + c_kappa + pow(c_kappa, 2.))/(pow(2.,
    1. + c_kappa) ∗ (c_kappa − 2.) ∗ (c_kappa − 1.));      /*
    C_κ = 2(1_{κ=2} + 1_{κ>2} c_κ)/m^2 (3d) */
const double CONST_Ckappa = 2. ∗ ( fmax (c_kappa − 2., std::numeric_limits <
    double > ::epsilon( ) ) > DBL_EPSILON ? ck : 1. ) / pow(CONST_Minf, 2.);      /*
    C_{α,γ} (3c) */
const double CONST_Calphagamma = ((CONST_ALPHA ∗ CONST_C ∗ gsl_sf_beta(2 −
    CONST_ALPHA, CONST_ALPHA) ∗ (fmax(1. − CONST_GAMMA,
    DBL_EPSILON) > DBL_EPSILON ? gsl_sf_beta_inc(2. − CONST_ALPHA, CONST_ALPHA,
    CONST_GAMMA) : 1.)/pow(CONST_Minf, CONST_ALPHA)) + CONST_Ckappa)/4.;      /*
    sample size */
const std
    ::size_t CONST_SAMPLE_SIZE = 30;      /*
        number of experiments */
    const int CONST_EXPERIMENTS = 25 · 10^4;
```

This code is used in chunk 22.

4.3 random number generators

define the STL *rng*( ) and GSL *rngtype* random number generators

7 ⟨ rngs 7 ⟩ ≡

```
/*
    random seed generator for the STL random number generator */
std::random_device randomseed;      /*
    Standard mersenne twister random number engine seeded with randomseed( ) */
std::mt19937_64 rng(randomseed( ));      /*
    define the GSL random number generator rngtype */
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 22.

4.4 the exponential function

compute e^x for the given x with error checking

8 ⟨ exponentialfunction 8 ⟩ ≡

```
static long double veldi(const long double &v)
{
  feclearexcept(FE_ALL_EXCEPT);
  const long double svar = expl(v);
  assert(fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) ≡ 0);
  return (svar);
}
```

This code is used in chunk 22.

4.5 the beta function

compute the logarithm of the (incomplete) beta function $\int_0^1 \mathbb{1}_{\{0<u\leq x\}} u^{a-1}(1-u)^{b-1}du$ using the Gauss hypergeometric function

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

9 ⟨ betafunction 9 ⟩ ≡

```
static long double betafunc(const double &a, const double &b)
{    /*
       the GSL incomplete beta function is normalised by the complete beta function   */
  assert(a > DBL_EPSILON);
  assert(b > DBL_EPSILON);    /*
       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⟩((1. − CONST_GAMMA > DBL_EPSILON ?
      gsl_sf_hyperg_2F1(a + b, 1., a + 1., CONST_GAMMA) : 1.));
  assert(f > DBL_EPSILON);    /*
       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 (1. − CONST_GAMMA > DBL_EPSILON ? (logl(f) + (static_cast⟨long
      double⟩((a * log(CONST_GAMMA)) + (b * log(1. − CONST_GAMMA)) − 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 22.

4.6 the counts constant

given merger sizes $k_1, \ldots, k_r$ compute $\prod_{j=2}^{m} \left(\sum_i \mathbb{1}_{\{k_i=j\}}\right)!$ (1)

⟨ countconstant 10 ⟩ ≡

```
static long double numbercollisions ( const std::vector < std::size_t > &__k )
{
  assert(__k[0] > 1);      /*
      get the number of (simultaneous) mergers r ∈ [4] */
  const int r = (__k[3] > 1 ? 4 : (__k[2] > 1 ? 3 : (__k[1] > 1 ? 2 : 1)));
  int l
  { }
  ;
  switch (r) {
  case 1:
    {
      l = 1;
      break;
    }
  case 2:
    {
      assert(__k[1] ≤ __k[0]);
      l = (__k[0] ≡ __k[1] ? 2 : 1);
      break;
    }
  case 3:
    {
      assert(__k[2] ≤ __k[1]);
      assert(__k[1] ≤ __k[0]);
      l = (__k[0] ≡ __k[2] ? 6 : (__k[0] ≡ __k[1] ? 2 : (__k[1] ≡ __k[2] ? 2 : 1)));
      break;
    }
  case 4:
    {
      assert(__k[3] ≤ __k[2]);
      assert(__k[2] ≤ __k[1]);
      assert(__k[1] ≤ __k[0]);
      l = (__k[0] ≡ __k[3] ? 24 : (__k[0] ≡ __k[2] ? 6 : ((__k[0] ≡ __k[1] ? (__k[2] ≡
          __k[3] ? 4 : 2) : (__k[1] ≡ __k[3] ? 6 : (__k[1] ≡ __k[2] ? 2 : (__k[2] ≡
          __k[3] ? 2 : 1)))))));
      break;
    }
  default: break;
  }
  assert(l > 0);      /*
      return ∏_{j=1}^{m} (∑_i 𝟙_{k_i=j})! */
  return static_cast⟨long double⟩(l);
}
```

This code is used in chunk 22.

4.7 the falling factorial

compute the logarithm of the falling factorial $(4)_m := \prod_{j=0}^{m-1}(4-j)$ with $(4)_0 := 1$

11 ⟨ falling 11 ⟩ ≡

```
static long double ff ( const std ::size_t &m )
    {
      assert(m < 5);
      return logl(static_cast⟨long double⟩(m > 0 ? (m < 2 ? 4 : (m < 3 ? 12 : 24)) : 1));
    }
```

This code is used in chunk 22.

4.8 the multinomial constant $\binom{m}{k_1 \ldots k_r \quad s}$

return the logarithm of the multinomial constant $\binom{m}{k_1 \ldots k_r \quad s}$ in (1) as $\log\Gamma(1+m) - \log\Gamma(1+k_1) - \mathbb{1}_{\{k_2 \geq 2\}} \log\Gamma(1+k_2) - \mathbb{1}_{\{k_3 \geq 2\}} \log\Gamma(1+k_3) - \mathbb{1}_{\{k_4 \geq 2\}} \log\Gamma(1+k_4) - \log\Gamma(1+m-\sum_i k_i)$

12 ⟨ multinomial 12 ⟩ ≡

```
static long double multinomialconstant ( const std ::size_t &m, const std::vector <
        std::size_t > &v__k )
{
  assert(m > 1);
  assert(v__k[0] > 1);
  long double svar = lgammal(static_cast⟨long double⟩(m + 1));
  svar −= lgammal(static_cast⟨long double⟩(v__k[1] + 1));
  svar −= (v__k[2] > 1 ? lgammal(static_cast⟨long double⟩(v__k[2]+1)) : (long
      double) 0.);
  svar −= (v__k[3] > 1 ? lgammal(static_cast⟨long double⟩(v__k[3]+1)) : (long
      double) 0.);
  svar −= (v__k[4] > 1 ? lgammal(static_cast⟨long double⟩(v__k[4]+1)) : (long
      double) 0.);
  svar −= lgammal(static_cast⟨long double⟩(1 + m − v__k[1] − v__k[2] −
      v__k[3] − v__k[4]));
  return svar;
}
```

This code is used in chunk 22.

4.9 the total merger rate $\lambda_{m;k_1,\ldots,k_r;s}$ (1)

given merger sizes the total merger rate $\lambda_{m;k_1,\ldots,k_r;s}$ (1)

13 $\langle\, \lambda_{m;k_1,\ldots,k_r;s}\ (1)\ 13\,\rangle \equiv$

```
static long double lambdankstotal ( const std ::size_t &m, const std::vector <
        std::size_t > &__k )
{
  assert(__k[0] > 1);
  assert(__k[3] ≤ __k[2]);
  assert(__k[2] ≤ __k[1]);
  assert(__k[1] ≤ __k[0]);
  const std
      ::size_t r = 1+(__k[1] > 1 ? 1 : 0)+(__k[2] > 1 ? 1 : 0)+(__k[3] > 1 ? 1 : 0);
  assert(r > 0);
  const std
      ::size_t k = __k[0] + __k[1] + __k[2] + __k[3];
  assert(k > 1);
  long double x
  { }
  ;
  long double y
  { }
  ;
  for (std::size_t ell = 0; ell ≤ MIN(m − k, 4 − r); ++ell) {      /*
        log (m k1...k4 s) § 4.8 */
    y = multinomialconstant(m, __k);      /*
        log (s ℓ) from (2) */
    y += static_cast⟨long double⟩(gsl_sf_lnchoose(m − k, ell));      /*
        (4)r+ℓ from (2); § 4.7 */
    y += ff(r + ell);      /*
        B(γ, Σj kj + ℓ − α, m − Σj kj − ℓ + α) from (2); § 4.5 */
    y += betafunc(static_cast⟨double⟩(k + ell) − CONST_ALPHA,
        static_cast⟨double⟩(m − k − ell) + CONST_ALPHA);      /*
        log 4^(Σj kj+ℓ) from (2) */
    y −= (static_cast⟨long double⟩(k + ell) ∗ logl(static_cast⟨long double⟩(4)));
      /*
        exp ( log (s ℓ) + log(4)r+ℓ + log B(γ, Σj kj + ℓ − α, m − Σj kj − ℓ + α) − log 4^(Σj kj+ℓ) )
        recalling (2); § 4.4 */
    x += veldi(y);
  }      /*
        ∏_{j=1}^{m} (Σi 1{ki=j})! from (1); § 4.6 */
  x /= numbercollisions(__k);      /*
      multiply with αc/m^α from (2) */
  x ∗= (CONST_ALPHA ∗ CONST_C/(powl(static_cast⟨long double⟩(CONST_Minf),
      static_cast⟨long double⟩(CONST_ALPHA))));      /*
      add Cκm(m − 1)/2 from (2) when Σj kj = 2 */
  x += (k < 3 ? static_cast⟨long double⟩((m ∗ (m − 1))/2) ∗ static_cast⟨long
      double⟩(CONST_Ckappa) : static_cast⟨long double⟩(0));
  x /= static_cast⟨long double⟩(CONST_Calphagamma);
  return (x);
}
```

This code is used in chunk 22.

4.10 the total jump rate out of m blocks

the total jump rate $\lambda_m = \sum_{(k_1,\ldots,k_4)\in\mathcal{K}_m} \lambda_{m;k_1,\ldots,k_4;s}$ out of m blocks where $\lambda_{m;k_1,\ldots,k_r;s}$ as in (1). Need to sum over all possible ordered merger sizes $\mathbf{k} \in \mathcal{K}_m$ given number of blocks m

14 ⟨ lambdan 14 ⟩ ≡

```
static void lambdan ( const std ::size_t &m, std::vector < long double > &v__lambdan )
{   /*
      m is current number of blocks */
  assert(m > 1);   /*
      __k will contain the merger sizes */
  std::vector < std::size_t > __k(4);
  for (std::size_t i = 2; i ≤ m; ++i) {
    __k = {i, 0, 0, 0};
    assert(std::accumulate(__k.begin( ), __k.end( ), 0) ≤ static_cast⟨int⟩(m));   /*
        § 4.9 */
    v__lambdan[m] += lambdankstotal(m, __k);
    if (MIN(i, m − i) > 1) {
      for (std::size_t j = 2; j ≤ MIN(i, m − i); ++j) {
        __k = {i, j, 0, 0};
        assert(__k[1] ≤ __k[0]);
        assert(std::accumulate(__k.begin( ), __k.end( ), 0) ≤ static_cast⟨int⟩(m));
        v__lambdan[m] += lambdankstotal(m, __k);
        if (MIN(j, m − i − j) > 1) {
          for (std::size_t k = 2; k ≤ MIN(j, m − i − j); ++k) {
            assert(k ≤ j);
            __k = {i, j, k, 0};
            assert(std::accumulate(__k.begin( ), __k.end( ), 0) ≤ static_cast⟨int⟩(m));
            v__lambdan[m] += lambdankstotal(m, __k);
            if (MIN(k, m − i − j − k) > 1) {
              for (std::size_t l = 2; l ≤ MIN(k, m − i − j − k); ++l) {
                __k = {i, j, k, l};
                assert(std::accumulate(__k.begin( ), __k.end( ),
                    0) ≤ static_cast⟨int⟩(m));
                v__lambdan[m] += lambdankstotal(m, __k);
                ;
              }
            }
          }
        }
      }
    }
  }
}
```

This code is used in chunk 22.

4.11 λ_m for $m = 2, 3, \ldots, n$

compute the total jump rate λ_m for $m = 2, 3, ..., n$ where n is sample size; for each $m = 2, 3, \ldots, n$ add up $\lambda_{m;k_1,\ldots,k_r;s}$ from (1) over the ordered merger sizes

15 ⟨ total jump rate all m 15 ⟩ ≡

```
static void lambdamallm ( std::vector < long double > &v__lambdam )
{    /*
       __lambdam stores the λ_m values; CONST_SAMPLE_SIZE § 4.2 */
  for (std::size_t n = 2; n ≤ CONST_SAMPLE_SIZE; ++n) {    /*
        lambdan § 4.10 */
    lambdan(n, v__lambdam);
  }
}
```

This code is used in chunk 22.

4.12 sample merger sizes

sample merger sizes by going through the ordered mergers until $u \leq F(\mathbf{k})$ where u is a random uniform and F is the sum of the merger probabilities up to and including $\mathbf{k}$

16 ⟨ sample **k** 16 ⟩ ≡

```
static void samplemergersizes ( const std ::size_t &m, const std::vector < long
        double > &__lambdan, std::vector < std::size_t > &__k )
{
  assert(__k.size( ) ≡ 4);
  std::size_t i = 1;
  std::size_t j
  { }
  ;
  std::size_t k
  { }
  ;
  std::size_t l
  { }
  ;
  long double F
  { }
  ;
  const double u = gsl_rng_uniform(rngtype);
  while ((i ≤ m) ∧ (u > F)) {
    ++i;
    __k = {i, 0, 0, 0};      /*
        lambdankstotal § 4.9 */
    F += lambdankstotal(m, __k)/__lambdan[m];
    if ((MIN(i, m − i) > 1) ∧ (u > F)) {
      j = 1;
      while ((j ≤ MIN(i, m − i)) ∧ (u > F)) {
        ++j;
        __k = {i, j, 0, 0};
        F += lambdankstotal(m, __k)/__lambdan[m];
        if ((MIN(j, m − i − j) > 1) ∧ (u > F)) {
          k = 1;
          while ((k ≤ MIN(j, m − i − j)) ∧ (u > F)) {
            ++k;
            __k = {i, j, k, 0};
            F += lambdankstotal(m, __k)/__lambdan[m];
            if ((MIN(k, m − i − j − k) > 1) ∧ (u > F)) {
              l = 1;
              while ((l ≤ MIN(k, m − i − j − k)) ∧ (u > F)) {
                ++l;
                __k = {i, j, k, l};
                F += lambdankstotal(m, __k)/__lambdan[m];
              }
            }
          }
        }
      }
    }
```

```
    }
  }
  assert(static_cast<std::size_t>(accumulate(__k.begin(), __k.end(), 0)) <= m);
}
```

This code is used in chunk 22.

4.13 merge blocks

merge blocks and record new block sizes given merger sizes

17 ⟨ merge blocks given merger sizes 17 ⟩ ≡

```
static void mergeblocks ( const std::vector < std::size_t > &__k, std::vector <
    std::size_t > &__blocks )
{   /*
      __k is merger sizes; __blocks the current block sizes */
  assert(__k[0] > 1);
  std::shuffle(__blocks.begin( ), __blocks.end( ), rng);
  const std
      ::size_t r = (__k[3] > 1 ? 4 : (__k[2] > 1 ? 3 : (__k[1] > 1 ? 2 : 1)));
      std::vector < std::size_t > newblocks(r);
  const std
      ::size_t m = __blocks.size( );
  for (std::size_t i = 0; i < r; ++i) {
    assert(__k[i] ≤ __blocks.size( ));
    newblocks[i] = static_cast⟨std::size_t⟩(std::accumulate(__blocks.rbegin( ),
        __blocks.rbegin( ) + __k[i], 0));
    assert(newblocks[i] > 1);
    __blocks.resize(__blocks.size( ) − __k[i]);
  }
  __blocks.insert(__blocks.end( ), newblocks.begin( ), newblocks.end( ));
  assert(__blocks.size( ) ≡ (m − (__k[0] + __k[1] + __k[2] + __k[3]) + r));
}
```

This code is used in chunk 22.

4.14 update branch lengths $\ell_i(n)$

given current block sizes __b update branch lengths $\ell_b(n) \leftarrow t + \ell_b(n)$ for $b = b_1, \ldots, b_m$ where t is a random exponential with rate λ_m

18 ⟨ update $\ell_i(n)$ 18 ⟩ ≡

```
static void updatebranchlengths (const double &t, const std::vector <
    std::size_t > &__b, std::vector < double > &__l ) {
for (const auto &b:__b)
{
  assert(b > 0);
  assert(b < CONST_SAMPLE_SIZE);
  __l[0] += t;
  __l[b] += t;
}
}
```

This code is used in chunk 22.

4.15 update relative branch lengths $r_i(n)$

given a realisation of branch lengths update estimate $\overline{\varrho}_i(n)$ of mean relative branch lengths

19 ⟨ update $\overline{\varrho}_i(n)$ 19 ⟩ ≡

```
static void updaterelativebranchlengths ( const std::vector < double > &__l, std::vector
    < double > &__r ) {      /*
     __l is vector of branch lengths; __r is vector of estimates ϱ̄_i(n)   */
assert(__l[0] > 0);
const double d = __l[0]; std::transform (__l.begin(), __l.end(), __r.begin(), __r.begin(),
        [&d](const auto &x, const auto &y)
{
  return y + (x/d);
}
) ; }
```

This code is used in chunk 22.

4.16 one experiment

generate one realisation of branch lengths - one experiment

20 ⟨ one experiment 20 ⟩ ≡

```
static void one ( std::vector < double > &__errs, const std::vector < long
        double > &__lambdam ) {
  std::vector < double > __ells(CONST_SAMPLE_SIZE);
  std::vector < std::size_t > __bees(CONST_SAMPLE_SIZE, 1);
  std::vector < std::size_t > __mergersizes(4);
  double t
  {}
  ;
  while (__bees.size() > 1) {      /*
        sample a random exponential with rate λ_m   */
    t = gsl_ran_exponential(rngtype, 1./__lambdam[__bees.size()]);      /*
        § 4.14 */
    updatebranchlengths(t, __bees, __ells);      /*
        sample merger sizes § 4.12   */
    samplemergersizes(__bees.size(), __lambdam, __mergersizes);      /*
        given merger sizes merge blocks § 4.13   */
    mergeblocks(__mergersizes, __bees);
  }      /*
        given branch lengths update relative branch lengths § 4.15   */
  updaterelativebranchlengths(__ells, __errs); }
```

This code is used in chunk 22.

4.17 estimate $\mathbb{E}[R_i(n)]$

obtain an estimate $\overline{\varrho}_i(n)$ of $\mathbb{E}[R_i(n)]$ (8) for the given number of experiments defined in § 4.2

21 ⟨ estimate 21 ⟩ ≡

```
static void estimate( ) {      /*
        § 4.2 */
    int M = CONST_EXPERIMENTS + 1; std::vector < long
        double > __lambdam(1 + CONST_SAMPLE_SIZE, 0); std::vector <
        double > __varrho(CONST_SAMPLE_SIZE, 0);      /*
         get the total rate § 4.11   */
    lambdamallm(__lambdam);      /*
         one § 4.16   */
    while (--M > 0) {
      one(__varrho, __lambdam);
    }
    for (const auto &v:__varrho)
    {
      std::cout ≪ v ≪ '\n';
    }
    }
```

This code is used in chunk 22.

4.18 the main module

the *main* function

```
22      /*
        § 4.1 */
  ⟨ includes 5 ⟩    /*
        § 4.2 */
      ⟨ constants 6 ⟩    /*
        § 4.3 */
      ⟨ rngs 7 ⟩    /*
        § 4.4 */
      ⟨ exponentialfunction 8 ⟩    /*
        § 4.5 */
      ⟨ betafunction 9 ⟩    /*
        § 4.6 */
      ⟨ countconstant 10 ⟩    /*
        § 4.7 */
      ⟨ falling 11 ⟩    /*
        § 4.8 */
      ⟨ multinomial 12 ⟩    /*
        § 4.9 */
      ⟨ λ_{m;k_1,...,k_r;s} (1) 13 ⟩    /*
        § 4.10 */
      ⟨ lambdan 14 ⟩    /*
        § 4.11 */
      ⟨ total jump rate all m 15 ⟩    /*
        § 4.12 */
      ⟨ sample k 16 ⟩    /*
        § 4.13 */
      ⟨ merge blocks given merger sizes 17 ⟩    /*
        § 4.14 */
      ⟨ update ℓ_i(n) 18 ⟩    /*
        § 4.15 */
      ⟨ update ϱ̄_i(n) 19 ⟩    /*
        § 4.16 */
      ⟨ one experiment 20 ⟩    /*
        § 4.17 */
      ⟨ estimate 21 ⟩
      int main(int argc, char *argv[])
      {    /*
             setup_rng § 4.3 */
        setup_rng(static_cast⟨std::size_t⟩(atoi(argv[1])));    /*
             estimate § 4.17 */
        estimate( );
        return GSL_SUCCESS;
      }
```

5 conclusions and bibliography

We estimate mean relative branch lengths $\mathbb{E}[R_i(n)]$ as predicted by the Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent. The Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent can be derived from a model of a diploid panmictic population of constant size evolving absent selfing, in a random environment, and according to randomly increased recruitment. The Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent extends the Ω-Beta$(2-\alpha, \alpha)$-coalescent derived from diploid panmictic populations [BLS15] and based on [Sch03]. The Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent extends the Ω-Beta$(2-\alpha, \alpha)$-coalescent by *(i)* including an atom at zero, *(ii)* extending the range of α from $[1, 2)$ to $(0, 2)$, and *(iii)* including a truncation $0 < \gamma \leq 1$. Moreover *(iv)*, the unit of time of the Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent is proportional to (at least) $N/\log N$ generations (where $2N$ is the population size); the unit of time for the Ω-Beta$(2-\alpha, \alpha)$-coalescent is proportional to $N^{\alpha-1}$ generations $(1 < \alpha < 2)$. The $N^{\alpha-1}$ unit of time might require strong assumptions regarding the population size or the mutation rate in order to recover the mutations in a given sample used to estimate α when α is estimated to be near 1.

References

[BLS15] https://dx.doi.org/10.1214/18-ejp175. 2018. 23, 0. Matthias Birkner and Huili Liu and Anja Sturm. Coalescent results for diploid exchangeable population models, Electronic Journal of Probability.

[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.

[F2025] Frederic Alberti and Matthias Birkner and Wai-Tong Louis Fan and John Wakeley. A conditional coalescent for diploid exchangeable population models given the pedigree. arXiv, 2025
https://dx.doi.org/10.48550/arXiv.2505.15481

[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

[S00] J Schweinsberg. Coalescents with simultaneous multiple collisions. *Electronic Journal of Probability*, 5:1–50, 2000.
https://dx.doi.org/10.1214/EJP.v5-68

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

Index

List of Refinements

⟨ $\lambda_{m;k_1,\ldots,k_r;s}$ (1) 13 ⟩ Used in chunk 22.
⟨ betafunction 9 ⟩ Used in chunk 22.
⟨ constants 6 ⟩ Used in chunk 22.
⟨ countconstant 10 ⟩ Used in chunk 22.
⟨ estimate 21 ⟩ Used in chunk 22.
⟨ exponentialfunction 8 ⟩ Used in chunk 22.
⟨ falling 11 ⟩ Used in chunk 22.
⟨ includes 5 ⟩ Used in chunk 22.
⟨ lambdan 14 ⟩ Used in chunk 22.
⟨ merge blocks given merger sizes 17 ⟩ Used in chunk 22.
⟨ multinomial 12 ⟩ Used in chunk 22.
⟨ one experiment 20 ⟩ Used in chunk 22.
⟨ rngs 7 ⟩ Used in chunk 22.
⟨ sample $\mathbf{k}$ 16 ⟩ Used in chunk 22.
⟨ total jump rate all m 15 ⟩ Used in chunk 22.
⟨ update $\ell_i(n)$ 18 ⟩ Used in chunk 22.
⟨ update $\overline{\varrho}_i(n)$ 19 ⟩ Used in chunk 22.