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 $\{\xi^n(t) : t \geq 0\}$ denote a coalescent, $\#A$ the number of elements in a given finite set A, n the number of diploid individuals sampled (so that $2n$ gene copies or chromosomes are sampled), $L_i(n) \equiv \int_0^{\tau(n)} \#\{\xi \in \xi^n(t) : \#\xi = i\}\, dt$ 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 $i \in \{1, 2, \ldots, 2n-1\}$; $R_i(n) \equiv L_i(n)/\sum_j L_j(n)$ for $i = 1, 2, \ldots, n-1$. Then $L_i(n)$ is interpreted as the random total length of branches supporting $i \in \{1, 2, \ldots, 2n-1\}$ leaves, with the length measured in coalescent time units. We then have $L(n) = L_1(n) + \cdots + L_{2n-1}(n)$. With this C++ code one estimates the functionals $\mathbb{E}[R_i(n)]$ when $\{\xi^n(t)\}$ is the Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$ coalescent where $0 < \gamma \leq 1$ and $0 < \alpha < 2$. The Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$ coalescents are a family of Ξ-coalescents where the asynchronous merger sizes are driven by a Beta$(\gamma, 2-\alpha, \alpha)$ measure with an atom at 0; the blocks participating in a merger are split into four groups (parent chromosomes) uniformly at random (with replacement) and the blocks in the same group are merged.

Contents

[1] beldon11@gmail.com

[2] compiled @ 3pm on Saturday 25th October, 2025
CTANGLE 4.12.1 (TeX Live 2025/Debian)
g++ (Debian 15.2.0-7) 15.2.0
kernel 6.16.12+deb14-amd64 GNU/Linux
GNU bash, version 5.3.3(1)-release (x86_64-pc-linux-gnu)
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
SpiX 1.3.0
GNU parallel 20240222
GNU Emacs 30.1

1 Copyright

2 compilation and output

This CWEB [KL94] document (the `.w` file) can be compiled with `cweave` to generate a `.tex` file, and with `ctangle` to generate a `.c` [KR88] C++ code file.

Use the shell tool `spix` on the script appearing before the preamble (the lines starting with %$); simply

`spix /path/to/the/sourcefile`

where `sourcefile` is the `.w` file

One may also copy the script into a file and run `parallel` [Tan11] :

`parallel --gnu -j1 :::: /path/to/scriptfile`

3 intro

Suppose a population is and evolves as in Definition

Definition 3.1 *Consider a diploid (each diploid individual carries two gene copies or chromosomes; each diploid individual can also be seen as a pair of chromosomes) panmictic population of constant size $2N$ diploid individuals. In any given generation we arbitrarily label each individual with a unique label, and form all possible $(2N-1)N$ (unordered) pairs of labels. Then we sample N pairs of labels independently and uniformly at random without replacement. The N parent pairs thus formed independently produce random numbers of diploid potential offspring according to some given law. Each offspring receives two chromosomes, one chromosome from each of its two parents, with each inherited chromosome sampled independently and uniformly at random from among the two parent chromosomes. If the total number of potential offspring is at least $2N$, we sample $2N$ of them uniformly at random without replacement to survive to maturity and replace the current individuals; otherwise we assume the population is unchanged over the generation (all the potential offspring perish before reaching maturity).*

Remark 3.2 *The mechanism described in Definition 3.1 is illustrated below, where $\{a,b\}$ denotes a diploid individual carrying gene copies a,b and $\{\{a,b\},\{c,d\}\}$ denotes a pair of parents. Here we have arbitrarily labelled the gene copies just for the sake of illustrating the evolution over one generation.*

stage	*individuals involved*
1	$\{\{a,b\},\{c,d\}\},\ldots,\{\{w,x\},\{y,z\}\}$ *: N parent pairs*
2	$\underbrace{\{a,d\},\{b,d\},\ldots,\{a,c\}}_{X_1},\ldots,\underbrace{\{w,y\},\ldots,\{x,z\}}_{X_N}$ *: $X_1+\cdots+X_N$ potential offspring*
3	$\{b,d\},\ldots,\{x,y\}$ *: $2N$ surviving offspring (whenever $X_1+\cdots+X_N\geq 2N$)*

In stage 1 above, the current $2N$ diploid individuals randomly form N pairs; in stage 2 the N pairs formed in stage 1 independently produce random numbers $X_1,\ldots,X_N$ of potential offspring, where each offspring receives one gene copy (or chromosome) from each of its two parents; in the third stage $2N$ of the $X_1+\cdots+X_N$ potential offspring (conditional on there being at least $2N$ of them) are sampled uniformly and without replacement to survive to maturity and replace the parents.

Write $R_i(n)\equiv L_i(n)/\sum_{j=1}^{2n-1}L_j(n)$. We approximate $\mathbb{E}\left[R_i^N(n)\right]$ when the sample comes from a finite diploid panmictic population evolving as in Definition 3.1.

Let $X_1,\ldots,X_N$ denote the random number of potential offspring produced by the N current parent pairs. The $X_1,\ldots,X_N$ are independent. Let X be independent of $X_1,\ldots,X_N$ and suppose

$$\mathbb{P}\left(X=k\right)=C\left(\frac{1}{k^a}-\frac{1}{(1+k)^a}\right),\quad k\in\{2,3,\ldots,\zeta(N)\} \tag{1}$$

with $a>0$ and we choose C such that $\mathbb{P}\left(X\in\{2,3,\ldots,\zeta(N)\}\right)=1$. Write $X\rhd L(a,\zeta(N))$ when X is distributed according to (1) for some given a and $\zeta(N)$. In any given generation the current individuals independently produce potential offspring according to (1); from the pool of potential offspring N of them are sampled uniformly at random and without replacement to survive and reach maturity and replace the current individuals. Note that (1) is an extension of [Sch03, Equation 11]. Suppose $X_1,\ldots,X_N$ are iid copies of X where $\mathbb{P}\left(X\rhd L(\alpha,\zeta(N))\right)=\varepsilon_N$, $\mathbb{P}\left(X\rhd L(\kappa,\zeta(N))\right)=1-\varepsilon_N$ where $1\leq\alpha<2$ and $\kappa\geq 2$ both fixed.

$r = \sum_i \mathbb{1}_{\{k_i \geq 2\}}$ and $k \sum_i \mathbb{1}_{\{k_i \geq 2\}} k_i$

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

$$\lambda'_{n;k_1,\ldots,k_r;s} = \mathbb{1}_{\{k=2\}} \frac{C_\kappa}{C_{\kappa,\alpha,\gamma}} + \frac{c\alpha}{m_\infty^\alpha C_{\kappa,\alpha,\gamma}} \sum_{\ell=0}^{s \wedge (4-r)} \binom{s}{\ell} \frac{(4)_{r+\ell}}{4^{k+\ell}} B(\gamma, k+\ell-\alpha, m-k-\ell+\alpha) \tag{2b}$$

$$m = \frac{1}{2}\left(2 + \frac{1 + 2^{1-\kappa}}{\kappa - 1}\right) \tag{2c}$$

$$\gamma = \mathbb{1}_{\left\{\frac{\zeta(N)}{N} \to K\right\}} + \mathbb{1}_{\left\{\frac{\zeta(N)}{N} \to \infty\right\}} \tag{2d}$$

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

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

where $\kappa + 2 < c_\kappa < \kappa^2$ when $\kappa > 2$. In (2b) $0 < \alpha < 2$.

The code follows in § 4.1 – § 4.20; we conclude in § 5. Comments within the code are in **this font and color**

4 code

4.1 the includes

the included libraries

5 ⟨ includes 5 ⟩ ≡

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <numeric>
#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 <unordered_set>
#include <forward_list>
#include <assert.h>
#include <math.h>
#include <fenv.h>
#include <unistd.h>
#include <limits>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_sf.h>
#include <boost/math/special_functions/beta.hpp>
#include <boost/math/special_functions/factorials.hpp>
#include "xindbetacoal_using_integer_partitions.hpp"
```

This code is used in chunk 24.

4.2 the random number generators

define the random number generators

```
std::random_device randomseed;
std::mt19937_64 rng(randomseed());

gsl_rng * rngtype ;
static void setup_rng(const 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) ;
}
```

6 ⟨ rngs 6 ⟩ ≡

```
/*
    obtain a seed for the random number engine */
std::random_device randomseed;      /*
    Standard mersenne twister random number engine */
std::mt19937_64 rng(randomseed( ));
gsl_rng * rngtype;
static void setup_rng(const 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.

4.3 exponential function

compute e^x checking for flows

```
long double veldi( const long double v )
{
feclearexcept(FE_ALL_EXCEPT);
const long double svar =  expl( v ) ;
return( fetestexcept( FE_OVERFLOW ) != 0 ? LDBL_MAX : (
fetestexcept(FE_UNDERFLOW) != 0 ? 0.0L : svar) ) ;

}
```

7 ⟨ e^x 7 ⟩ ≡

```
long double veldi(const long double v)
{
  feclearexcept(FE_ALL_EXCEPT);
  const long double svar = expl(v);
  return (fetestexcept(FE_OVERFLOW) ≠ 0 ? LDBL_MAX : (fetestexcept(FE_UNDERFLOW) ≠ 0 ?
      0.0L : svar));
}
```

This code is used in chunk 24.

4.4 the logarithm of the beta function

the logarithm of the beta$(x, a, b) \equiv \int_0^x t^{a-1}(1-t)^{b-1}dt$
the GSL incomplete beta function is normalised by the complete beta function
the standard way would be $gsl\_sf\_beta\_inc(a, b, x) * gsl\_sf\_beta(a, b)$
$\mathbb{1}_{\{x<1\}}(\log f + a\log x + b\log(1-x) - \log a) + \mathbb{1}_{\{x=1\}}(\log\Gamma(a) + \log\Gamma(b) - \log\Gamma(a+b))$

```
1  static long double betafunc( const double x, const double a, const double b )
2  {
3  assert( x > 0.);
4  assert( a > 0.);
5  assert( b > 0.);

6  const long  double f = static_cast<long double>( (x < 1. ? gsl_sf_hyperg_2F1(a
7  + b, 1, a+1.,x) : 1.) );
8  assert( lessthan(0.0L, f) );

9  return( x < 1. ?  (logl( f ) +  ( static_cast<long double>( (a*log(x)) + (b *
10 log(1-x)) - log(a) ) ) ) : (lgammal( static_cast<long double>(a)) + lgammal(
11 static_cast<long double>(b) ) - lgammal( static_cast<long double>(a + b) ) ) \
 + ↪ );
12 }
```

8 ⟨ log Beta(x, a, b) 8 ⟩ ≡

```
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 */
      /*
        0 < x ≤ 1 is the cutoff point */
  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.));      /*
        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 24.

4.5 number of collisions

compute $\prod_{j=2}^{n} \left(\sum_i \mathbb{1}_{\{k_i\}} = j\right)!$ where k_i are the number of blocks assigned to each of 4 groups (uniformly at random with replacement), and n is the current number of blocks, so that $2 \leq \sum_i k_i \leq n$

```
static long double numbercollisions( const std::vector<unsigned int>& __k )
{
assert( std::all_of( __k.cbegin(), __k.cend(), [](const auto x){return x > 1;})
);

double l {} ;
switch( __k.size() ){
case 1 : {
l = 1;  break ; }
case 2 : {
assert( __k[0] <= __k[1]);
l = (__k[0] == __k[1] ? 2 : 1);
break ; }
case 3 : {
assert( __k[1] <= __k[2]);
assert( __k[0] <= __k[1]);
l = ( __k[0] == __k[2] ? 6 : ( __k[0] == __k[1] ? 2 : ( __k[1] == __k[2] ? 2 :
1)));
break ; }
case 4 : {
assert( __k[2] <= __k[3]);
assert( __k[1] <= __k[2]);
assert( __k[0] <= __k[1]);
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 static_cast<long double>(l) ;
}
```

9 ⟨ product of collision numbers 9 ⟩ ≡

```
static long double numbercollisions ( const std::vector < unsigned int > &___k ) {
assert ( std::all_of (___k.cbegin( ),___k.cend( ), [](const auto x)
{
  return x > 1;
}
) ) ;
double l
{ }
;
switch (___k.size( )) {
case 1:
```

```
    {
      l = 1;
      break;
    }
  case 2:
    {
      assert(___k[0] ≤ ___k[1]);
      l = (___k[0] ≡ ___k[1] ? 2 : 1);
      break;
    }
  case 3:
    {
      assert(___k[1] ≤ ___k[2]);
      assert(___k[0] ≤ ___k[1]);
      l = (___k[0] ≡ ___k[2] ? 6 : (___k[0] ≡ ___k[1] ? 2 : (___k[1] ≡ ___k[2] ? 2 : 1)));
      break;
    }
  case 4:
    {
      assert(___k[2] ≤ ___k[3]);
      assert(___k[1] ≤ ___k[2]);
      assert(___k[0] ≤ ___k[1]);
      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 static_cast⟨long double⟩(l); }
```

This code is used in chunk 24.

4.6 descending factorial

descending factorial $(x)_m \equiv x(x-1)\cdots(x-m+1)$ and $(x)_0 \equiv 1$; return $\log(x)_m$

```
long double ff( const long double x, const long double m)
{
long double f = 1.0L;
long double j = 0.0L;
while(j < m){
f *= (x - j);
++j ;
}
assert(f > LDBL_EPSILON );
return logl(f) ;
}
```

10 $\langle\, (x)_m \;10\,\rangle \equiv$

```
long double ff(const long double x,const long double m)
{
  long double f = 1.0L;
  long double j = 0.0L;
  while (j < m) {
    f *= (x - j);
    ++j;
  }
  assert(f > LDBL_EPSILON);
  return logl(f);
}
```

This code is used in chunk 24.

4.7 multinomial coefficient

compute

$$\log \binom{n}{k_1 \cdots k_r s} - \log \prod_{j=2}^{n} \left(\sum_i \mathbb{1}_{\{k_i = j\}} \right)!$$

```
long double multinomialconstant( const unsigned int m, const
std::vector<unsigned int>& v_k)
{
const long double s = lgammal( static_cast<long double>(1 +  m -
std::accumulate( v__k.cbegin(), v__k.cend(), 0)) ) ;
long double d =   static_cast<long double>(std::accumulate( v__k.cbegin(),
v__k.cend(), 0., [](long double a, const auto x){return a +
lgammal(static_cast<long double>(x+1)); })) ;
return ( lgammal(static_cast<long double>(m + 1)) - d - s -
logl(numbercollisions(v_k)) ) ;
}
```

11 ⟨ $\log \binom{n}{k_1 \cdots k_r s}$ 11 ⟩ ≡

```
long double multinomialconstant (const unsigned int m, const std::vector < unsigned
    int > &v___k ) {
const long double s = lgammal(static_cast⟨long
    double⟩(1 + m - std::accumulate(v___k.cbegin( ), v___k.cend( ), 0)));
long double d = static_cast⟨long double⟩ ( std::accumulate (v___k.cbegin( ),
        v___k.cend( ), 0.0L, [](long double a, const auto x)
{
  return a + lgammal(static_cast⟨long double⟩(x + 1));
}
) ) ;     /*
    § 4.5 */
return (lgammal(static_cast⟨long double⟩(m + 1)) - d - s - logl(numbercollisions(v___k)));
    }
```

This code is used in chunk 24.

4.8 $\lambda_{n;k_1,\ldots,k_r;s}$

compute $\lambda_{n;k_1,\ldots,k_r;s}$ (2a)

12 ⟨ $\lambda_{n;k_1,\ldots,k_r;s}$ 12 ⟩ ≡

```
long double lambdanks (const unsigned int m, const std::vector < unsigned
    int > &v___k ) { const unsigned int r = v___k.size( );
const unsigned int k = std::accumulate(v___k.cbegin( ), v___k.cend( ), 0);
assert(k > 1);
assert(k ≤ m);
const unsigned int s = m − k; auto logchoose = [](const unsigned int x, const unsigned
        int y)
{
  return (lgammal(static_cast⟨long double⟩(x + 1)) − lgammal(static_cast⟨long
      double⟩(y + 1)) − lgammal(static_cast⟨long double⟩(x − y + 1)));
}
;
long double l = 0.0L;
for (std::size_t ell = 0; ell ≤ (s < 4 − r ? s : 4 − r); ++ell) {
  l += veldi(logchoose(s, ell) + ff(4.0L, static_cast⟨long double⟩(r + ell)) + betafunc(GAMMA,
      static_cast⟨long double⟩(k + ell) − ALPHA,
      static_cast⟨long double⟩(m − k − ell) + ALPHA) − (logl(4.0L) ∗ static_cast⟨long
      double⟩(k + ell)));
}
l ∗= (ALPHA ∗ C_C);
l /= (CKAG ∗ powl(MM, ALPHA));
l += (k < 3 ? CKAPPA/CKAG : 0.0L);
assert(l > LDBL_EPSILON);      /∗
    § 4.7 ∗/
return (veldi(logl(l) + multinomialconstant(m, v___k))); }
```

This code is used in chunk 24.

4.9 structure `RM_t`

for storing the integer partitions (ordered merger sizes)

13 ⟨ structure rmt 13 ⟩ ≡

```
struct RM_t {
std::vector < std::vector < unsigned int >> r___p
{ }
; } ;
```

This code is used in chunk 24.

4.10 generate integer partitions

generate ordered integer partitions of at least 2 elements and summing to *myInt*

14 ⟨ size 2 or larger integer partitions 14 ⟩ ≡

```
void GenPartitions (const unsigned int m, std::vector < long double > &v___lambdan,
        std::vector < RM_t > &rates_partitions, std::vector < std::vector
        < long double >> &rates_for_sorting, const unsigned int
        myInt, const unsigned int PartitionSize, unsigned int MinVal,
        unsigned int MaxVal ) { long double rate
{ }
; std::vector < unsigned int > partition(PartitionSize);
unsigned int idx_Last = PartitionSize − 1;
unsigned int idx_Dec = idx_Last;
unsigned int idx_Spill = 0;
unsigned int idx_SpillPrev;
unsigned int LeftRemain = myInt − MaxVal − (idx_Dec − 1) ∗ MinVal;
partition[idx_Dec] = MaxVal + 1;
do {
  unsigned int val_Dec = partition[idx_Dec] − 1;
  partition[idx_Dec] = val_Dec;
  idx_SpillPrev = idx_Spill;
  idx_Spill = idx_Dec − 1;
  while (LeftRemain > val_Dec) {
    partition[idx_Spill−−] = val_Dec;
    LeftRemain −= val_Dec − MinVal;
  }
  partition[idx_Spill] = LeftRemain;
  char a = (idx_Spill) ? ∼((−3 ≫ (LeftRemain − MinVal)) ≪ 2) : 11;
  char b = (−3 ≫ (val_Dec − LeftRemain));
  switch (a & b) {
  case 1: case 2: case 3: idx_Dec = idx_Spill;
    LeftRemain = 1 + (idx_Spill − idx_Dec + 1) ∗ MinVal;
    break;
  case 5:
    for (++idx_Dec, LeftRemain = (idx_Dec − idx_Spill) ∗ val_Dec;
        (idx_Dec ≤ idx_Last) ∧ (partition[idx_Dec] ≤ MinVal);
        idx_Dec++) LeftRemain += partition[idx_Dec];
    LeftRemain += 1 + (idx_Spill − idx_Dec + 1) ∗ MinVal;
    break;
  case 6: case 7: case 11: idx_Dec = idx_Spill + 1;
    LeftRemain += 1 + (idx_Spill − idx_Dec + 1) ∗ MinVal;
    break;
  case 9:
    for (++idx_Dec, LeftRemain = idx_Dec ∗ val_Dec;
        (idx_Dec ≤ idx_Last) ∧ (partition[idx_Dec] ≤ (val_Dec + 1));
        idx_Dec++) LeftRemain += partition[idx_Dec];
    LeftRemain += 1 − (idx_Dec − 1) ∗ MinVal;
    break;
  case 10:
    for (LeftRemain += idx_Spill ∗ MinVal + (idx_Dec −
        idx_Spill) ∗ val_Dec + 1, ++idx_Dec;
        (idx_Dec ≤ idx_Last) ∧ (partition[idx_Dec] ≤ (val_Dec − 1));
        idx_Dec++) LeftRemain += partition[idx_Dec];
```

```
    LeftRemain -= (idx_Dec - 1) * MinVal;
    break;
  }
  while (idx_Spill > idx_SpillPrev) partition[--idx_Spill] = MinVal;
  const unsigned int sama_summa = static_cast<unsigned
      int>(std::accumulate(partition.cbegin(), partition.cend(), 0)) == myInt;
  switch (sama_summa) {
  case 1:
    {     /*
            § 4.8 */
      rate = lambdanks(m, partition);
      v___lambdan[m] += rate;
      (rates_partitions[m]).r___p.push_back(partition);
      rates_for_sorting[m].push_back(rate);
      break;
    }
  case 0: break;
  default: break;
  }
 } while (idx_Dec <= idx_Last); }
```

This code is used in chunk 24.

4.11 all merger sizes when m blocks

15 ⟨ mergers for m blocks 15 ⟩ ≡

```
static void allmergers_for_m_blocks (const unsigned int m, std::vector < long
    double > &v___lambdan, std::vector < RM_t > &rates_partitions, std::vector <
    std::vector < long double >> &rates_for_sorting ) { std::vector < unsigned
    int > onemerger(1);
long double lm
{ }
;
for (unsigned int s = 2; s ≤ m; ++s) {
  onemerger[0] = s;      /*
      § 4.8 */
  lm = lambdanks(m, onemerger);
  v___lambdan[m] += lm;
  rates_partitions[m].r___p.push_back(onemerger);
  rates_for_sorting[m].push_back(lm);
}
if (m > 3) {      /*
      get the partitions of size at least 2 */
  unsigned int number_groups
  { }
  ;     /*
      k is the sum of the merger sizes, partitions  */
  for (unsigned int k = 4; k ≤ m; ++k) {
    number_groups = k/2 > 4 ? 4 : k/2;
    for (unsigned int s = 2; s ≤ number_groups; ++s) {      /*
        § 4.10 */
      GenPartitions(m, v___lambdan, rates_partitions, rates_for_sorting, k, s, 2, k−(2∗(s−1)));
    }
  }
}
assert(rates_for_sorting[m].size( ) > LDBL_EPSILON); }
```

This code is used in chunk 24.

4.12 order the rates

order the rates (2a) in descending order for generating CMFs for sampling mergers

16 ⟨ put the rates in descending order 16 ⟩ ≡

```
static void order_rates ( const std::vector < long double > &v___rates_for_sorting,
    std::vector < unsigned int > &v___indx ) {
assert(v___rates_for_sorting.size( ) > 0);
v___indx.clear( );
v___indx.resize(v___rates_for_sorting.size( ));
std::iota(v___indx.begin( ), v___indx.end( ), 0);
std::stable_sort (v___indx.begin( ), v___indx.end( ), [&v___rates_for_sorting](const
        unsigned int x, const unsigned int y)
{
  return v___rates_for_sorting[x] > v___rates_for_sorting[y];
}
) ; }
```

This code is used in chunk 24.

4.13 compute a CMF

generate a cumulative mass function from sorted rates (2a) for sampling mergers

17 ⟨ cmf 17 ⟩ ≡

```
static void generate_cmf_n_blocks (const long double lambdan, const std::vector < long
    double > &v___rates, const std::vector < unsigned int > &v___indx, std::vector
    < long double > &v___cmf )
{
  v___cmf.clear( );
  v___cmf.resize(v___rates.size( ));
  v___cmf[0] = v___rates[v___indx[0]]/lambdan;
  for (unsigned int i = 1; i < v___indx.size( ); ++i) {
    v___cmf[i] = v___cmf[i − 1] + (v___rates[v___indx[i]]/lambdan);
  }
  assert(fabsl(v___cmf.back( ) − 1.0 L) < 1.0 · 10^−9 L);
}
```

This code is used in chunk 24.

4.14 get all merger sizes

generate all ordered mergers

18 ⟨ all mergers 18 ⟩ ≡

```
static void allmergers ( std::vector < long double > &v___lambdan, std::vector
    < RM_t > &a___rates_partitions, std::vector < std::vector < long
    double >> &a___rates_for_sorting, std::vector < std::vector < unsigned
    int >> &a___indx, std::vector < std::vector < long double >> &a___cmfs )
{
  for (unsigned int n = 2; n ≤ 2 * SAMPLE_SIZE; ++n) {      /*
      § 4.11 */
    allmergers_for_m_blocks(n, v___lambdan, a___rates_partitions, a___rates_for_sorting);
    assert(a___rates_for_sorting[n].size( ) > 0);
    assert(v___lambdan[n] > LDBL_EPSILON);      /*
      § 4.12 */
    order_rates(a___rates_for_sorting[n], a___indx[n]);      /*
      § 4.13 */
    generate_cmf_n_blocks(v___lambdan[n], a___rates_for_sorting[n], a___indx[n],
      a___cmfs[n]);
  }
}
```

This code is used in chunk 24.

4.15 udpate the lengths ℓ_i

19 ⟨ update realisations of $L_i(n)$ 19 ⟩ ≡

```
void update_lengths (const long double l, const std::vector < unsigned
    int > &v___tree, std::vector < long double > &v___lengths )
{
  assert(l > LDBL_EPSILON);
  const long double t = -logl(static_cast⟨long double⟩(1. - gsl_rng_uniform(rngtype)))/l;
  for (unsigned int i = 0; i < v___lengths.size( ); ++i) {
    v___lengths[0] += t;
    v___lengths[v___tree[i]] += t;
  }
}
```

This code is used in chunk 24.

4.16 sample merger

sample mergers

20 ⟨ pick merger 20 ⟩ ≡

```
unsigned int pick_merger ( const std::vector < long double > &v___cmf )
{
  unsigned int j = 0;
  const long double u = static_cast⟨long double⟩(gsl_rng_uniform(rngtype));
  assert(u ≤ 1.0L);
  while (u > v___cmf[j]) {
    ++j;
  }
  return j;
}
```

This code is used in chunk 24.

4.17 given mergers update tree

given merger sizes merge blocks and update tree

21 ⟨ merge blocks and update tree 21 ⟩ ≡

```
void update_tree ( std::vector < unsigned int > &v___tree, const std::vector < unsigned
    int > &merger_sizes ) {
  assert(merger_sizes.size( ) > 0);
  assert(static_cast⟨unsigned int⟩(std::accumulate(merger_sizes.cbegin( ),
      merger_sizes.cend( ), 0)) ≤ v___tree.size( ));
  std::shuffle(v___tree.begin( ), v___tree.end( ), rng);
  std::vector < unsigned int > newblocks
  { }
  ;
  newblocks.clear( );
  newblocks.reserve(merger_sizes.size( ));
  assert(newblocks.size( ) < 1);
  for (const auto &s:merger_sizes)
  {
    newblocks.push_back(std::accumulate(v___tree.crbegin( ), v___tree.crbegin( ) + s, 0));
    v___tree.resize(v___tree.size( ) − s);
  }
  v___tree.insert(v___tree.end( ), newblocks.cbegin( ), newblocks.cend( )); }
```

This code is used in chunk 24.

4.18 one experiment

one experiment

22 ⟨ one realisation 22 ⟩ ≡

```
void one_experiment ( const std::vector < long double > &v___lambdan, std::vector <
        long double > &v___lengths, const std::vector < std::vector < long
        double >> &a___cmfs, const std::vector < RM_t > &a___partitions,
        const std::vector < std::vector < unsigned int >> &a___indx, std::vector
        < long double > &v___ri ) {
  std::vector < unsigned int > v___tree(2 * SAMPLE_SIZE, 1);
  std::fill(v___lengths.begin( ), v___lengths.end( ), 0.0L);
  unsigned int m
  { }
  ;
  while (v___tree.size( ) > 1) {      /*
        § 4.15 */
    update_lengths(v___lambdan[v___tree.size( )], v___tree, v___lengths);      /*
        § 4.16 */
    m = pick_merger(a___cmfs[v___tree.size( )]);      /*
        § 4.17 */
    update_tree(v___tree,
        a___partitions[v___tree.size( )].r___p[a___indx[v___tree.size( )][m]]);
  }
  assert(v___tree.back( ) ≡ (2 * SAMPLE_SIZE));
  assert(v___lengths[0] > LDBL_EPSILON);
  const long double d = v___lengths[0];
  std::transform (v___lengths.cbegin( ), v___lengths.cend( ), v___ri.begin( ),
          v___ri.begin( ), [&d](const auto x, const auto y)
  {
    return y + (x/d);
  }
  ) ; }
```

This code is used in chunk 24.

4.19 approximate $\mathbb{E}[R_i(n)]$

23 ⟨ go ahead – approximate $\mathbb{E}[R_i(n)]$ 23 ⟩ ≡

```
void approximate( ) {
    std::vector < long double > v___lambdan((2 * SAMPLE_SIZE) + 1);      /*
        § 4.9 */
    std::vector < RM_t > a___parts ((2 * SAMPLE_SIZE) + 1, RM_t { } ) ;
    std::vector < std::vector < long double >> a_rates_sorting ((2 * SAMPLE_SIZE) + 1,
        std::vector < long double > { } ) ;
    std::vector < std::vector < long double >> a___cmfs ((2 * SAMPLE_SIZE) + 1,
        std::vector < long double > { } ) ;
    std::vector < std::vector < unsigned int >> a___indx ((2 * SAMPLE_SIZE) + 1,
        std::vector < unsigned int > { } ) ;
    std::vector < long double > v___lengths(2 * SAMPLE_SIZE);
    std::vector < long double > v___ri(2 * SAMPLE_SIZE);      /*
        § 4.14 */
    allmergers(v___lambdan, a___parts, a_rates_sorting, a___indx, a___cmfs);
    int r = EXPERIMENTS + 1;
    while (--r > 0)      /*
        § 4.18 */
    {
      one_experiment(v___lambdan, v___lengths, a___cmfs, a___parts, a___indx, v___ri);
    }
    for (const auto &z:v___ri)
    {
      std::cout ≪ z ≪ '\n';
    }
    }
```

This code is used in chunk 24.

4.20 main

the *main* module

```
24    /*
      § 4.1 */
  ⟨ includes 5 ⟩    /*
      § 4.2 */
    ⟨ rngs 6 ⟩    /*
      § 4.3 */
    ⟨ e^x 7 ⟩    /*
      § 4.4 */
    ⟨ log Beta(x, a, b) 8 ⟩    /*
      § 4.5 */
    ⟨ product of collision numbers 9 ⟩    /*
      § 4.6 */
    ⟨ (x)_m 10 ⟩    /*
      § 4.7 */
    ⟨ log (n choose k_1 ··· k_r s) 11 ⟩    /*
      § 4.8 */
    ⟨ λ_{n;k_1,...,k_r;s} 12 ⟩    /*
      § 4.9 */
    ⟨ structure rmt 13 ⟩    /*
      § 4.10 */
    ⟨ size 2 or larger integer partitions 14 ⟩    /*
      § 4.11 */
    ⟨ mergers for m blocks 15 ⟩    /*
      § 4.12 */
    ⟨ put the rates in descending order 16 ⟩    /*
      § 4.13 */
    ⟨ cmf 17 ⟩    /*
      § 4.14 */
    ⟨ all mergers 18 ⟩    /*
      § 4.15 */
    ⟨ update realisations of L_i(n) 19 ⟩    /*
      § 4.16 */
    ⟨ pick merger 20 ⟩    /*
      § 4.17 */
    ⟨ merge blocks and update tree 21 ⟩    /*
      § 4.18 */
    ⟨ one realisation 22 ⟩    /*
      § 4.19 */
    ⟨ go ahead – approximate E[R_i(n)] 23 ⟩
    int main(int argc, const char *argv[])
    {    /*
          § 4.2 */
      setup_rng(static_cast⟨std::size_t⟩(atoi(argv[1])));    /*
          § 4.19 */
      approximate( );
      gsl_rng_free(rngtype);
      return 0;
    }
```

5 conclusion and bibliography

we approximate the functionals $\mathbb{E}[R_i(n)]$ for $i = 1, 2, \ldots, 2n-1$ when the coalescent is the Ω-δ_0-Beta$(\gamma, 2-\alpha, \alpha)$ coalescent with transition rates as in (2a). Figure 1 holds an example

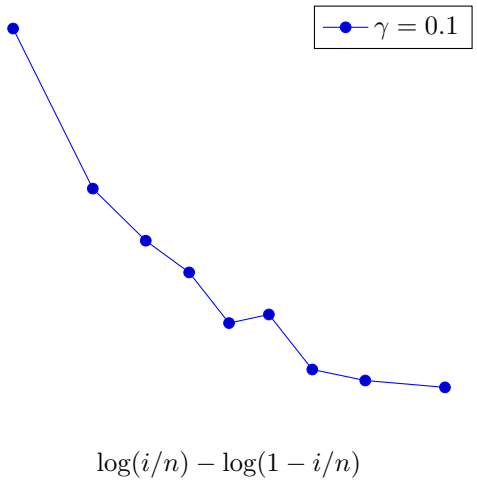

Figure 1: *An example approximation of* $\mathbb{E}[R_i(n)]$ *for the given parameter values and graphed as logits against* $\log(i/n) - \log(1 - i/n)$ *for* $i = 1, 2, \ldots, 2n-1$ *where* n *is sample size*

References

[D2024] Diamantidis, Dimitrios and Fan, Wai-Tong (Louis) and Birkner, Matthias and Wakeley, John. Bursts of coalescence within population pedigrees whenever big families occur. Genetics Volume 227, February 2024.
https://dx.doi.org/10.1093/genetics/iyae030.

[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

⟨ $(x)_m$ 10 ⟩ Used in chunk 24.
⟨ $\lambda_{n;k_1,\ldots,k_r;s}$ 12 ⟩ Used in chunk 24.
⟨ $\log \binom{n}{k_1 \cdots k_r s}$ 11 ⟩ Used in chunk 24.
⟨ $\log \mathrm{Beta}(x, a, b)$ 8 ⟩ Used in chunk 24.
⟨ e^x 7 ⟩ Used in chunk 24.
⟨ all mergers 18 ⟩ Used in chunk 24.
⟨ cmf 17 ⟩ Used in chunk 24.
⟨ go ahead – approximate $\mathbb{E}[R_i(n)]$ 23 ⟩ Used in chunk 24.
⟨ includes 5 ⟩ Used in chunk 24.
⟨ merge blocks and update tree 21 ⟩ Used in chunk 24.
⟨ mergers for m blocks 15 ⟩ Used in chunk 24.
⟨ one realisation 22 ⟩ Used in chunk 24.
⟨ pick merger 20 ⟩ Used in chunk 24.
⟨ product of collision numbers 9 ⟩ Used in chunk 24.
⟨ put the rates in descending order 16 ⟩ Used in chunk 24.
⟨ rngs 6 ⟩ Used in chunk 24.
⟨ size 2 or larger integer partitions 14 ⟩ Used in chunk 24.
⟨ structure rmt 13 ⟩ Used in chunk 24.
⟨ update realisations of $L_i(n)$ 19 ⟩ Used in chunk 24.