

Gene genealogies in diploid populations evolving according to sweepstakes reproduction

— approximating $\mathbb{E}[R_i(n)]$ for the time-changed Ω - δ_0 -Beta($\gamma, 2 - \alpha, \alpha$)-coalescent

BJARKI ELDON ¹² 

Let $\#A$ denote the number of elements in a finite set A . For a given coalescent $\{\xi^n\}$ write $L_i(n) \equiv \int_0^{\tau(n)} \#\{\xi \in \xi^n(t) : \#\xi = i\} dt$ and $L(n) \equiv \int_0^{\tau(n)} \#\xi^n(t) dt$ where $\tau(n) \equiv \inf\{t \geq 0 : \#\xi^n(t) = 1\}$. Then $L(n) = L_1(n) + \dots + L_{n-1}(n)$. Write $R_i(n) \equiv L_i(n)/L(n)$ for $i = 1, 2, \dots, n-1$. With this C++ simulation code we approximate $\mathbb{E}[R_i(n)]$ when $\{\xi^n(G)\} \equiv \{\xi^n(G(t)) : t \geq 0\}$ is the time-changed continuous-time Ω - δ_0 -Beta($\gamma, 2 - \alpha, \alpha$) coalescent with $0 < \alpha < 2$ and time-change function $G(t)$.

Contents

1	Copyright	2
2	compilation and output	3
3	intro	4
4	code	5
4.1	includes	6
4.2	constants	7
4.3	random number generators	8
4.4	compute e^x checking for under and overflow	9
4.5	number of collisions	10
4.6	multinomial constant	11
4.7	the (incomplete) beta function	12
4.8	read merger sizes	13
4.9	$\lambda_{n;k_1,\dots,k_r;s}$	14
4.10	rates for a given number of blocks	15
4.11	order the rates in descending order	16
4.12	generate cmf	17
4.13	compute all merger rates	18
4.14	get time until next merger	19
4.15	update lengths	20
4.16	update block sizes	21
4.17	one experiment	22
4.18	approximate	24
4.19	main	25
5	conclusion and references	26

¹beldon11@gmail.com

²compiled @ 2:18pm on Monday 20th October, 2025

CTANGLE 4.12.1 (TeX Live 2025/Debian)

g++ (Debian 15.2.0-4) 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)

L^AT_EX This is LuaHBTeX, Version 1.22.0 (TeX Live 2025/Debian) Development id: 7673

written using GNU Emacs 30.1

1 Copyright

Copyright © 2025 Bjarki Eldon

This document and any source code it contains is distributed under the terms of the GNU General Public Licence (version ≥ 3). You should have received a copy of the licence along with this file (see file COPYING).

The source codes described in this document are free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This document and the code it contains is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this file (see COPYING). If not, see <http://www.gnu.org/licenses/>.

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

Write $\mathbb{N} \equiv \{1, 2, \dots\}$, $[n] = \{1, 2, \dots, n\}$, $]n] \equiv \{2, 3, \dots, n\}$ for all $n \in \mathbb{N}$. The Ω - δ_0 -Beta($\gamma, 2 - \alpha, \alpha$)-coalescent is a continuous-time Ξ -coalescent with transition rates

$$\begin{aligned} \lambda_{n;k_1, \dots, k_r; s} &= \mathbb{1}_{\{r=1, k_1=2\}} \frac{C_\kappa}{C_{\alpha, \gamma}} + \\ &+ \frac{c\alpha}{C_{\alpha, \gamma} \mathfrak{m}^\alpha} \sum_{\ell=0}^{s \wedge (4-r)} \binom{s}{\ell} \frac{(4)_{r+\ell}}{4^{k+\ell}} \int_0^1 \mathbb{1}_{\{0 < t \leq \gamma\}} t^{k+\ell-\alpha-1} (1-t)^{n+\alpha-k-\ell-1} dt \end{aligned} \quad (1)$$

where $0 < \gamma \leq 1$, $B(p, a, b) \equiv \int_0^1 \mathbb{1}_{\{0 < t \leq p\}} t^{a-1} (1-t)^{b-1} dt$, and

$$\begin{aligned} C_\kappa &= \frac{2}{4\mathfrak{m}^2} \left(\mathbb{1}_{\{\kappa=2\}} + \mathbb{1}_{\{\kappa>2\}} \frac{c_\kappa}{2\kappa(\kappa-2)(\kappa-1)} \right) \\ C_{\kappa, \alpha, \gamma} &= C_\kappa + \frac{c\alpha}{4\mathfrak{m}^\alpha} B(\gamma, 2 - \alpha, \alpha) \\ \mathfrak{m} &= \mathbb{1}_{\{\kappa=2\}} \left(2\frac{\pi^2}{3} - 3 \right) + \mathbb{1}_{\{\kappa>2\}} \left(1 + 2^{\kappa-1} \frac{\kappa}{\kappa-1} \right) \end{aligned}$$

In (1) $n \geq 2$, $k_1, \dots, k_r \in]n]$ and $2 \leq \sum_i k_i \leq n$ for all $r \in [4]$, $s \equiv n - \sum_i k_i$.

Here we consider a time-changed version of the Ω - δ_0 -Beta($\gamma, 2 - \alpha, \alpha$)-coalescent corresponding to exponential population growth where the time-change function $G(t) = \int_0^t e^{\rho s} ds$ for some fixed $\rho \geq 0$.

Let $\#A$ denote the number of elements in a finite set A . For a given coalescent $\{\xi^n\}$ write $L_i(n) \equiv \int_0^{\tau(n)} \# \{\xi \in \xi^n(t) : \#\xi = i\} dt$ and $L(n) \equiv \int_0^{\tau(n)} \#\xi^n(t) dt$ where $\tau(n) \equiv \inf \{t \geq 0 : \#\xi^n(t) = 1\}$. Then $L(n) = L_1(n) + \dots + L_{n-1}(n)$. Write $R_i(n) \equiv L_i(n)/L(n)$ for $i = 1, 2, \dots, n-1$. With this C++ code we use simulations to approximate $\mathbb{E}[R_i(n)]$

The code follows in § 4.1–§ 4.19, we conclude in § 5. Comments within the code in **this font and color**

4 code

4.1 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>
```

This code is used in chunk 23.

4.2 constants

the global constants

```

6  < constants 6 > ≡      /*
    Model 0:  $1 \leq \alpha < 2$  : epsilon =  $cN^{**}(\alpha - 2)(1(\kappa > 2) + (\kappa = 2)\log N)$ ; */      /*
    Model 1:  $\alpha = 1$  : epsilon = constant
     $0 < \alpha < 1$  : epsilon = constant * N * (cutoff)**(alpha - 1) */
    const double dalpha = 0.01;
    const long double ALPHA = static_cast<long double>(dalpha);
    const long double KAPPA = 2.0L;
    const long double RHO = 0.0L;
    const double dgamma = 0.1;
    const long double C_C = 1.0L;      /*
    when KAPPA > 2: const long double MM =  $1 + 2^\kappa * \kappa / (\kappa - 1)$ ; */      /*
    when KAPPA = 2: */
    const long double MM =  $(2.0_L * M\_PII * M\_PII / 3.0_L) - 3.0_L$ ;
    const long double CA =  $((KAPPA + 2.0_L) + (KAPPA * KAPPA)) / 2.0_L$ ;
    const long double CB =  $powl(2.0_L, KAPPA) * ((KAPPA - 2.0_L) * (KAPPA - 1.0_L))$ ;
    const long double CKAPPA =  $2.0_L * (KAPPA > 2.0_L ? CA / CB : 1.0_L) / (MM * MM)$ ;
    const long double GAMMA = static_cast<long double>(dgamma);
    const long double BETA = static_cast<long double>(gsl_sf_beta(2. - dalpha,
        dalpha) * (dgamma < 1. ? gsl_sf_beta_inc(2. - dalpha, dalpha, dgamma) : 1.));
    const long double CKAG =  $(CKAPPA + ((ALPHA * C\_C * powl(2.0_L, ALPHA)) * BETA / powl(MM,$ 
        ALPHA / 4.0L));      /*
    SAMPLE_SIZE is number of diploid individuals sampled */
    const unsigned int SAMPLE_SIZE = 100;
    const int EXPERIMENTS = 2500;

```

This code is used in chunk 23.

4.3 random number generators

the random number generators

```
7  ⟨rngs 7⟩ ≡  
    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 23.

4.4 compute e^x checking for under and overflow

compute e^x checking for under and overflow

```
1 long double veldi( const long double v )
2 {
3     feclearexcept(FE_ALL_EXCEPT);

4     const long double svar =  expl( v ) ;

5     return( fetestexcept( FE_OVERFLOW ) != 0 ? LDBL_MAX : (
6     fetestexcept(FE_UNDERFLOW) != 0 ? 0.0L : svar) ) ;
7 }
```

8 $\langle e^x$ with checks 8 $\rangle \equiv$

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

This code is used in chunk 23.

4.5 number of collisions

get the factorial number of collisions $\prod_{j=2}^n (\sum_i \mathbb{1}_{\{k_i=j\}})!$

9 $\langle \text{numbercollisions } 9 \rangle \equiv$

```
static long double numbercollisions ( const std::vector < unsigned int > &__k ) {    /*
    __k is in ascending order */
    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 > DBL_EPSILON );
    return static_cast<long double>(l); }
```

This code is used in chunk 23.

4.6 multinomial constant

compute the log of the multinomial constant

$$\binom{m}{k_1 \dots k_r s} \frac{1}{\prod_{j=2}^n (\sum_i \mathbb{1}_{\{k_i=j\}})!}$$

10 \langle multinomial constant 10 $\rangle \equiv$

```

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)));
    const long double d = static_cast<long double> ( std::accumulate
        (v__k.cbegin(), v__k.cend(), 0.0_L, [](long double a, const auto x)
        {
            return a + lgammal(static_cast<long double>(x + 1));
        }
        ) );    /*
        numbercollisions § 4.5 */
    return (lgammal(static_cast<long double>(m + 1)) - d - s - logl(numbercollisions(v__k)));
}

```

This code is used in chunk 23.

4.7 the (incomplete) beta function

return the logarithm of the (incomplete) beta function; when complete it is of course $\log \Gamma(a) + \log \Gamma(b) - \log \Gamma(a + b)$

11 \langle the beta function 11 $\rangle \equiv$

```
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) */
    const long double f = static_cast<long double>((x < 1. ? gsl_sf_hyperg_2F1(a + b, 1,
        a + 1., x) : 1.));
    assert(LDBL_EPSILON < f);    /*
        return  $\mathbb{1}_{\{x < 1\}}(\log f + a \log x + b \log(1 - x) - \log a) + \mathbb{1}_{\{x = 1\}}(\log \Gamma(a) + \log \Gamma(b) - \log \Gamma(a + b))$ 
        */
    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 23.

4.8 read merger sizes

read in merger sizes summing to a given number; the merger sizes are available in the files

`Q_<m>_.txt`

where `m` is the given number

12 `<readmersizes 12> ≡`

```
static void readmersizes (const unsigned int n, const unsigned int j, std::vector <
    unsigned int > &v__mers) {
    std::ifstream f("Q_" + std::to_string(n) + "_.txt");
    std::string line {}
    ;
    v__mers.clear();
    for (unsigned int i = 0; std::getline (f, line) & i < j; ++i) { if (i ≥ j - 1) {
        std::stringstream (line) ;
        v__mers = std::vector < unsigned int > ( std::istream_iterator < unsigned int > (ss),
            {} ) ; } }
    assert(v__mers.size() > 0);
    assert ( std::all_of (v__mers.cbegin(), v__mers.cend(), [](const auto &x)
    {
        return x > 1;
    }
    ) ) ;
    f.close(); }
```

This code is used in chunk 23.

4.9 $\lambda_{n;k_1,\dots,k_r;s}$

compute $\lambda_{n;k_1,\dots,k_r;s}$ (1)

```

13  $\langle \lambda_{n;k_1,\dots,k_r;s} \rangle_{13} \equiv$ 
    long double lambdanks (const unsigned int m, const std::vector < unsigned
        int > &v___k ) { /*
        r is number of simultaneous mergers */
    const unsigned int r = v___k.size(); /*
        k = k1 + ... + kr */
    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; /*
        log  $\binom{x}{y} = \log \Gamma(x+1) - \log \Gamma(y+1) - \log \Gamma(x-y+1)$  */
    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; /*
        (4)p ≡ 4(4-1) ... (4-p+1) and (4)0 ≡ 1 */
    auto ff = [] (const unsigned int p)
    {
        return (p > 0 ? static_cast<long double>(boost::math::falling_factorial(4.0, p)) : 1.0L);
    }
    ;
    for (std::size_t ell = 0; ell ≤ (s < 4 - r ? s : 4 - r); ++ell) {
        l += veldi(logchoose(s, ell) + ff(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) * powl(2.0L, ALPHA);
    l /= (CKAG * powl(MM, ALPHA));
    l += (k < 3 ? CKAPPA/CKAG : 0.0L);
    assert(l > 0.0L);
    return (veldi(logl(l) + multinomialconstant(m, v___k))); }

```

This code is used in chunk 23.

4.10 rates for a given number of blocks

read in all the possible merger sizes when a given number of blocks and compute the rate (1)

14 \langle rates when given number of blocks 14 $\rangle \equiv$

```

void allrates_when_n_blocks (const unsigned int n, std::vector < long
    double > &v__lambdan, std::vector < long double > &v__rates ) {
    std::vector < unsigned int > v__k
    { }
    ;
    v__k.clear(); std::string line { }
    ;
    std::ifstream f("Q_" + std::to_string(n) + "_.txt");
    assert(f.is_open());
    std::stringstream ss
    { }
    ;
    long double l
    { }
    ;
    v__rates.clear();
    while ( std::getline (f, line) ) {
        ss = std::stringstream ( line );
        v__k = std::vector < unsigned int > ( std::istream_iterator < unsigned int > (ss),
        { } );
        assert(v__k.size() > 0);
        assert ( std::all_of (v__k.cbegin(), v__k.cend(), [](const auto &x)
        {
            return x > 1;
        }
        ) ); /*
        compute  $\lambda_{n;k_1,\dots,k_r;s}$  (1) § 4.9 */
        l = lambdanks(n, v__k);
        v__lambdan[n] += l;
        v__rates.push_back(l); }
    f.close(); }

```

This code is used in chunk 23.

4.11 order the rates in descending order

order the rates in descending order for generating the cmf for sampling merger sizes

```
1 static void order_rates( const std::vector< long double > &
2 v__rates_for_sorting, std::vector< unsigned int > & v__indx )
3 {
4     assert( v__rates_for_sorting.size() > 0 ) ;
5     v__indx.clear();
6     v__indx.resize( v__rates_for_sorting.size() ) ;
7     std::iota( v__indx.begin(), v__indx.end(), 0 ) ;
8     std::stable_sort(v__indx.begin(), v__indx.end(), [&v__rates_for_sorting](const
9 unsigned int x, const unsigned int y){return v__rates_for_sorting[x] >
10 v__rates_for_sorting[y];});
11 }
```

15 $\langle \text{order rates 15} \rangle \equiv$

```
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 23.

4.12 generate cmf

generate cmf for a given set of merger sizes

```

1  static void generate_cmf_n_blocks( const long double lambdan,  const
2  std::vector<long double>& v__rates, const std::vector<unsigned int> & v__indx,
3  std::vector<long double>& v__cmf )
4  {
5  v__cmf.clear();
6  v__cmf.resize( v__rates.size() );
7  v__cmf[0] = v__rates[v__indx[0]] / lambdan ;
8  for( unsigned int i = 1; i < v__indx.size(); ++i)
9  {
10 v__cmf[i] = v__cmf[ i-1] + ( v__rates[ v__indx[i] ] / lambdan );
11 }
12 assert( fabs( v__cmf.back() - 1.0L ) < 1.0e-9L ) ;
13 }

```

16 \langle generate cmf 16 $\rangle \equiv$

```

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 )
{
    /*
        lambdan is the total merger rate */
    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(fabs(v__cmf.back() - 1.0L) < 1.0 · 10-9L);
}

```

This code is used in chunk 23.

4.13 compute all merger rates

compute the rate (1) for all possible mergers

17 \langle all rates 17 $\rangle \equiv$

```
void allrates ( std::vector < long double > &v__lambdan, std::vector < std::vector <
  unsigned int >> &a__indx, std::vector < std::vector < long double >> &a__cmfs ) {
  std::vector < std::vector < long double >> a__rates (SAMPLE_SIZE + SAMPLE_SIZE + 1,
  std::vector < long double > { } ) ;    /*
  SAMPLE_SIZE § 4.2 is number of diploid individuals sampled */
  for (unsigned int m = 2; m ≤ SAMPLE_SIZE + SAMPLE_SIZE; ++m) {    /*
    § 4.10 */
    allrates_when_n_blocks(m, v__lambdan, a__rates[m]);
    assert(v__lambdan[m] > LDBL_EPSILON);    /*
    § 4.11 */
    order_rates(a__rates[m], a__indx[m]);    /*
    § 4.12 */
    generate_cmfs_n_blocks(v__lambdan[m], a__rates[m], a__indx[m], a__cmfs[m]);
    a__rates[m].resize(0);
    std::vector < long double > ( ).swap(a__rates[m]); }    /*
    clean up */
    for (auto &v:a__rates) { v.resize(0); std::vector < long double > ( ).swap(v); }
    std::vector < std::vector < long double >> ( ).swap(a__rates); }
```

This code is used in chunk 23.

4.14 get time until next merger

get the time

$$t = \mathbb{1}_{\{\rho > 0\}} \frac{1}{\rho} \log(1 - \rho \log(1 - U)e^{-\rho s}/\lambda_n) + \mathbb{1}_{\{\rho = 0\}} \text{Exp}(\lambda_n)$$

where U is a standard random uniform and s is the sum of the previous times

```

1 long double newtime(const long double lambdab, const long double oldtime)
2 {
3     return (RHO > LDBL_EPSILON ? log1pl( -(RHO * expl(- RHO * oldtime) / lambdab) \
+     ↪ *
4     logl(static_cast<long double>(gsl_rng_uniform_pos(rngtype)))))/RHO :
5     static_cast<long double>(gsl_ran_exponential(rngtype, 1./lambdab))) ;
6 }

```

18 $\langle \text{new time 18} \rangle \equiv$

```

    long double newtime(const long double lambdab, const long double oldtime)
    {
        /*
         * log1p(x) is log(1 + x) */
        return (RHO > LDBL_EPSILON ? log1pl(-(RHO * expl(-RHO * oldtime)/lambdab) *
            logl(static_cast<long double>(gsl_rng_uniform_pos(rngtype)))))/RHO :
            static_cast<long double>(gsl_ran_exponential(rngtype, 1./lambdab)));
    }

```

This code is used in chunk 23.

4.15 update lengths

update functionals $L_i(n)$ given current block sizes

19 $\langle \text{update lengths } 19 \rangle \equiv$

```
void update_lengths ( const std::vector < unsigned int > &v___tree, std::vector < long
    double > &v___lengths, const long double t )
{
    for (unsigned int i = 0; i < v___lengths.size(); ++i) {
        v___lengths[0] += t;
        assert(v___tree[i] > 0);
        assert(v___tree[i] < SAMPLE_SIZE + SAMPLE_SIZE);
        v___lengths[v___tree[i]] += t;
    }
}
```

This code is used in chunk 23.

4.16 update block sizes

merge blocks and record the continuing ones

```
1 void update_tree( std::vector<unsigned int> & v__tree, const
2 std::vector<unsigned int>& merger_sizes )
3 {
4     assert( merger_sizes.size() > 0 ) ;
5     assert( static_cast<unsigned int>(std::accumulate( merger_sizes.cbegin(),
6 merger_sizes.cend(),0)) <= v__tree.size() );
7     std::shuffle( v__tree.begin(), v__tree.end(), rng );
8     std::vector<unsigned int > newblocks {};
9     newblocks.clear();
10    newblocks.reserve( merger_sizes.size() ) ;
11    assert( newblocks.size() < 1 ) ;
12    for( const auto &s : merger_sizes){
13        newblocks.push_back( std::accumulate( v__tree.cbegin(), v__tree.cbegin() + \
+ ↪ s,
14 0));
15    v__tree.resize( v__tree.size() - s ) ; }

16    v__tree.insert( v__tree.end(), newblocks.cbegin(), newblocks.cend() );
17 }
```

20 <update block sizes 20> ≡

```
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.cbegin(), v__tree.cbegin() + 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 23.

4.17 one experiment

```

1 void one_experiment( const std::vector<long double>& v__lambdan,
2 std::vector<long double>& v__lengths, const std::vector< std::vector< long
3 double >> & a__cmfs, const std::vector< std::vector< unsigned int> > &
4 a__indx, std::vector<long double>& v__ri )
5 {
6 std::vector< unsigned int> v__tree( 2*SAMPLE_SIZE, 1);
7 std::fill( v__lengths.begin(), v__lengths.end(), 0.0L) ;
8 unsigned int merger_lina {} ;
9 std::vector< unsigned int > merger_sizes {} ;
10 long double otimi {};
11 long double it {};

12 auto mlina = [](const std::vector<long double>& f)
13 {
14 unsigned int j = 0;
15 const long double u = static_cast<long double>(gsl_rng_uniform(rngtype));
16 assert( u <= 1.0L );

17 while( u > f[j]){++j;}

18 return j ;};

19 merger_sizes.reserve(4);
20 while( v__tree.size() > 1)
21 {
22 it = newtime( v__lambdan[v__tree.size()], otimi);
23 otimi += it ;
24 update_lengths(v__tree, v__lengths, it);
25 merger_lina = mlina( a__cmfs[v__tree.size()] ) ;
26 readmergersizes( v__tree.size(), a__indx[v__tree.size()][merger_lina]+1,
27 merger_sizes);
28 update_tree( v__tree, merger_sizes ) ;
29 }
30 assert( v__tree.back() == (2*SAMPLE_SIZE));

31 assert( v__lengths[0] > LDBL_EPSILON) ;
32 const long double d = v__lengths[0];
33 std::transform( v__lengths.cbegin(), v__lengths.cend(), v__ri.begin(),
34 v__ri.begin(), [&d](const auto x, const auto y){return y + (x/d);});
35 }

```

21 \langle one experiment 21 $\rangle \equiv$

```

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 < 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 merger_lina
{}

```

```

; std::vector < unsigned int > merger_sizes
{}
;
long double otimi
{}
;
long double it
{}
; auto mlina = [] ( const std::vector < long double > &f )
{
    unsigned int j = 0;
    const long double u = static_cast<long double>(gsl_rng_uniform(rngtype));
    assert(u ≤ 1.0L);
    while (u > f[j]) {
        ++j;
    }
    return j;
}
;
merger_sizes.reserve(4);
while (v__tree.size() > 1) { /*
    § 4.14 */
    it = newtime(v__lambdan[v__tree.size()], otimi);
    otimi += it; /*
    § 4.15 */
    update_lengths(v__tree, v__lengths, it);
    merger_lina = mlina(a__cmfs[v__tree.size()]); /*
    § 4.8 */
    readmergersizes(v__tree.size(), a__indx[v__tree.size()][merger_lina] + 1,
        merger_sizes); /*
    § 4.16 */
    update_tree(v__tree, merger_sizes);
}
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 23.

4.18 approximate

```

1 void approximate()
2 {
3     std::vector<long double> v__lambdan ((2*SAMPLE_SIZE) + 1) ;

4     std::vector< std::vector< long double > > a__cmfs ( (2*SAMPLE_SIZE) + 1,
5     std::vector< long double > {} ) ;
6     std::vector< std::vector< unsigned int > > a__indx ( (2*SAMPLE_SIZE) + 1,
7     std::vector< unsigned int > {} ) ;
8     std::vector<long double> v__lengths (2*SAMPLE_SIZE) ;
9     std::vector< long double> v__ri (2*SAMPLE_SIZE) ;

10    allrates(v__lambdan, a__indx, a__cmfs);

11    int r = EXPERIMENTS + 1;

12    while( --r > 0)
13    {one_experiment(v__lambdan, v__lengths, a__cmfs, a__indx, v__ri) ; }

14    for( const auto &z:v__ri)
15    {std::cout << z << '\n';}
16    }

```

22 \langle go ahead – approximate $\mathbb{E}[R_i(n)]$ 22 $\rangle \equiv$

```

    void approximate() {
        std::vector< long double > v__lambdan((2 * SAMPLE_SIZE) + 1);
        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); /*
        § 4.13 */
        allrates(v__lambdan, a__indx, a__cmfs);
        int r = EXPERIMENTS + 1;
        while ( --r > 0) { /*
        § 4.17 */
            one_experiment(v__lambdan, v__lengths, a__cmfs, a__indx, v__ri);
        }
        for (const auto &z:v__ri)
        {
            std::cout << z << '\n';
        }
    }

```

This code is used in chunk 23.

4.19 main

the *main* module

```
23      /*
      § 4.1 */
      <includes 5> /*
      § 4.2 */
      <constants 6> /*
      § 4.3 */
      <rngs 7> /*
      § 4.4 */
      < $e^x$  with checks 8> /*
      § 4.5 */
      <numbercollisions 9> /*
      § 4.6 */
      <multinomial constant 10> /*
      § 4.7 */
      <the beta function 11> /*
      § 4.8 */
      <readmersizes 12> /*
      § 4.9 */
      < $\lambda_{n;k_1,\dots,k_r;s}$  13> /*
      § 4.10 */
      <rates when given number of blocks 14> /*
      § 4.11 */
      <order rates 15> /*
      § 4.12 */
      <generate cmf 16> /*
      § 4.13 */
      <all rates 17> /*
      § 4.14 */
      <new time 18> /*
      § 4.15 */
      <update lengths 19> /*
      § 4.16 */
      <update block sizes 20> /*
      § 4.17 */
      <one experiment 21> /*
      § 4.18 */
      <go ahead – approximate  $\mathbb{E}[R_i(n)]$  22>
      int main(int argc, const char *argv[])
      {
      /*
      § 4.3 */
      setup_rng(static_cast<std::size_t>(atoi(argv[1]))); /*
      § 4.18 */
      approximate(); /*
      rngtype § 4.3 */
      gsl_rng_free(rngtype);
      return GSL_SUCCESS;
      }
```

5 conclusion and references

Figure 1 records an example approximation of $\mathbb{E}[R_i(n)]$ given the parameter values in § 4.2

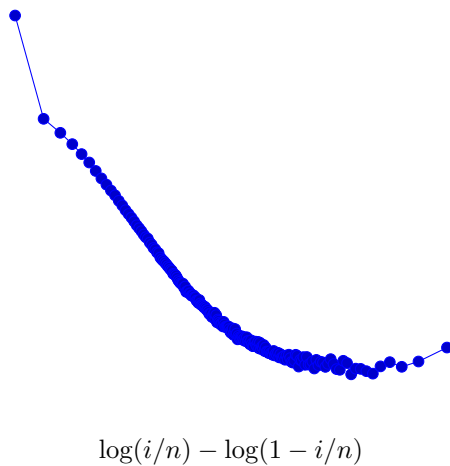


Figure 1: *An example approximation of $\mathbb{E}[R_i(n)]$ for the given parameter values and graphed as logits against $\log(i/n) - \log(1 - i/n)$ for $i = 1, 2, \dots, n - 1$ where n is sample size*

6 bibliography

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>.
- [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](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

__k: 9.
a: 10, 11.
a__cmfs: 17, 21, 22.
a__indx: 17, 21, 22.
a__rates: 17.
accumulate: 10, 13, 20.
all_of: 9, 12, 14.
allrates: 17, 22.
allrates_when_n_blocks: 14, 17.
ALPHA: 6, 13.
approximate: 22, 23.
argc: 23.
argv: 23.
assert: 9, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21.
atoi: 23.
b: 11.
back: 16, 21.
begin: 15, 20, 21.
BETA: 6.
betafunc: 11, 13.
boost: 13.
C_C: 6, 13.
CA: 6.
CB: 6.
cbegin: 9, 10, 12, 13, 14, 20, 21.
cend: 9, 10, 12, 13, 14, 20, 21.
CKAG: 6, 13.
CKAPPA: 6, 13.
clear: 12, 14, 15, 16, 20.
close: 12, 14.
cout: 22.
crbegin: 20.
d: 10, 21.
dalpha: 6.
DBL_EPSILON: 9.
dgamma: 6.
ell: 13.
end: 15, 20, 21.
EXPERIMENTS: 6, 22.
expl: 8, 18.
f: 11.
fabsl: 16.
falling_factorial: 13.
FE_ALL_EXCEPT: 8.
FE_OVERFLOW: 8.
FE_UNDERFLOW: 8.
feclearexcept: 8.
fetestexcept: 8.
ff: 13.
fill: 21.
GAMMA: 6, 13.
generate_cmf_n_blocks: 16, 17.
getline: 12, 14.
gsl_ran_exponential: 18.
gsl_rng: 7.
gsl_rng_alloc: 7.
gsl_rng_default: 7.
gsl_rng_env_setup: 7.
gsl_rng_free: 23.
gsl_rng_set: 7.
gsl_rng_type: 7.
gsl_rng_uniform: 21.
gsl_rng_uniform_pos: 18.
gsl_sf_beta: 6, 11.
gsl_sf_beta_inc: 6, 11.
gsl_sf_hyperg_2F1: 11.
GSL_SUCCESS: 23.
i: 12, 16, 19.
ifstream: 12, 14.
insert: 20.
iota: 15.
is_open: 14.
istream_iterator: 12, 14.
it: 21.
j: 12, 21.
k: 13.
KAPPA: 6.
l: 9, 13, 14.
lambdab: 18.
lambdan: 16.
lambdanks: 13, 14.
LDBL_EPSILON: 11, 17, 18, 21.
LDBL_MAX: 8.
lgammal: 10, 11, 13.
log: 11.
logchoose: 13.
logl: 10, 11, 13, 18.
log1p: 18.
log1pl: 18.
long: 10.
m: 10, 13, 17.
M_PI: 6.
main: 23.
math: 13.
merger_lina: 21.
merger_sizes: 20, 21.
mlina: 21.
MM: 6, 13.
mt19937_64: 7.
multinomialconstant: 10, 13.
n: 12, 14.
newblocks: 20.
newtime: 18, 21.

numbercollisions: 9, 10.
oldtime: 18.
one_experiment: 21, 22.
order_rates: 15, 17.
otimi: 21.
p: 13.
powl: 6, 13.
push_back: 14, 20.
r: 13, 22.
random_device: 7.
randomseed: 7.
readmersizes: 12, 21.
reserve: 20, 21.
resize: 15, 16, 17, 20.
RHO: 6, 18.
rng: 7, 20.
rngtype: 7, 18, 21, 23.
s: 7, 10, 13, 20.
SAMPLE_SIZE: 6, 17, 19, 21, 22.
setup_rng: 7, 23.
shuffle: 20.
size: 9, 12, 13, 14, 15, 16, 19, 20, 21.
ss: 12, 14.
stable_sort: 15.
std: 7, 9, 10, 12, 13, 14, 15, 16, 17, 19,
20, 21, 22, 23.
string: 12, 14.
stringstream: 12, 14.
svar: 8.
swap: 17.
T: 7.
t: 19.
to_string: 12, 14.
transform: 21.
u: 21.
unsigned: 13.
update_lengths: 19, 21.
update_tree: 20, 21.
v: 8, 17.
v__cmf: 16.
v__indx: 15, 16.
v__k: 10, 13, 14.
v__lambdan: 14, 17, 21, 22.
v__lengths: 19, 21, 22.
v__mergers: 12.
v__rates: 14, 16.
v__rates_for_sorting: 15.
v__ri: 21, 22.
v__tree: 19, 20, 21.
vector: 9, 10, 12, 13, 14, 15, 16, 17, 19,
20, 21, 22.
veldi: 8, 13.
x: 9, 10, 11, 12, 13, 14, 15, 21.
y: 13, 15, 21.

z: 22.

List of Refinements

- $\langle \lambda_{n;k_1,\dots,k_r;s} \ 13 \rangle$ Used in chunk 23.
- $\langle e^x$ with checks 8 \rangle Used in chunk 23.
- \langle all rates 17 \rangle Used in chunk 23.
- \langle constants 6 \rangle Used in chunk 23.
- \langle generate cmf 16 \rangle Used in chunk 23.
- \langle go ahead – approximate $\mathbb{E}[R_i(n)]$ 22 \rangle Used in chunk 23.
- \langle includes 5 \rangle Used in chunk 23.
- \langle multinomial constant 10 \rangle Used in chunk 23.
- \langle new time 18 \rangle Used in chunk 23.
- \langle numbercollisions 9 \rangle Used in chunk 23.
- \langle one experiment 21 \rangle Used in chunk 23.
- \langle order rates 15 \rangle Used in chunk 23.
- \langle rates when given number of blocks 14 \rangle Used in chunk 23.
- \langle readmergersizes 12 \rangle Used in chunk 23.
- \langle rngs 7 \rangle Used in chunk 23.
- \langle the beta function 11 \rangle Used in chunk 23.
- \langle update block sizes 20 \rangle Used in chunk 23.
- \langle update lengths 19 \rangle Used in chunk 23.