

Gene genealogies in diploid populations evolving according to sweepstakes reproduction

— approximating $\mathbb{E}[R_i(n)]$ for the time-changed $\Omega\text{-}\delta_0\text{-Poisson-Dirichlet}(\alpha, 0)$ -coalescent

BJARKI ELDON^{1,2} 

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 $\Omega\text{-}\delta_0\text{-Poisson-Dirichlet}(\alpha, 0)$ coalescent with $0 < \alpha < 1$ 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 random number generators	7
4.3 e^x with checks	8
4.4 $\lambda_{n;k_1, \dots, k_r;s}$	9
4.5 generate group sizes	10
4.6 group sizes up to a given number	12
4.7 write partitions to files	13
4.8 partitions when given number of blocks	14
4.9 all partitions	15
4.10 read in partitions	16
4.11 split partitions	17
4.12 split groups in partition	19
4.13 merge blocks	20
4.14 sample time and partitions until a merger occurs	21
4.15 one experiment	23
4.16 approximate $\mathbb{E}[R_i(n)]$	25
4.17 main	26
5 conclusions and bibliography	27

¹beldon11@gmail.com

²compiled @ 6:11pm on Tuesday 21st 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)

LATEX 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

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

One samples the Ω - δ_0 -Poisson-Dirichlet($\alpha, 0$)-coalescent by sampling group sizes $2 \leq k_1, \dots, k_r \leq n$ with $\sum_i k_i \leq n$, $s = n - \sum_i k_i$, according to

$$\begin{aligned}\lambda_{n;k_1,\dots,k_r;s} &= \mathbb{1}_{\{r=1, k_1=2\}} \frac{C_\kappa}{C_\kappa + c(1-\alpha)} + \frac{cp_{n;k_1,\dots,k_r;s}}{C_\kappa + c(1-\alpha)} \\ C_\kappa &= \frac{2}{m_\infty^2} \left(\mathbb{1}_{\{\kappa=2\}} + \mathbb{1}_{\{\kappa>2\}} \frac{c_\kappa}{2^\kappa (\kappa-2)(\kappa-1)} \right) \\ p_{n;k_1,\dots,k_r;s} &= \frac{\alpha^{r+s-1} (r+s-1)!}{(n-1)!} \prod_{i=1}^r (k_i - 1 - \alpha)_{k_i-1}\end{aligned}\quad (1)$$

where $2 + \kappa < c_\kappa < \kappa^2$ when $\kappa > 2$. The blocks in each group are then split among four subgroups independently and uniformly at random, and the blocks in the same subgroup are merged.

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 $\mathbb{E}[R_i(n)]$ when the coalescent is the time-changed continuous-time Ω - δ_0 -Poisson-Dirichlet($\alpha, 0$)-coalescent with time-change function $G(t) = \int_0^t e^{\rho s} ds$ for a given fixed $\rho \geq 0$.

The code follows in § 4.1–§ 4.17, we conclude in § 5. Comments within the code in **this font and color**

4 code

4.1 includes

the included libraries and header file with the global constants

```
5 < includes 5 > ≡  
#include <iostream>  
#include <cstdlib>  
#include <iterator>  
#include <random>  
#include <fstream>  
#include <iomanip>  
#include <vector>  
#include <numeric>  
#include <functional>  
#include <algorithm>  
#include <cmath>  
#include <unordered_map>  
#include <assert.h>  
#include <float.h>  
#include <fenv.h>  
#include <gsl/gsl_rng.h>  
#include <gsl/gsl_randist.h>  
#include <gsl/gsl_math.h>  
#include <boost/math/special_functions/factorials.hpp>  
#include "headerfile.hpp"
```

This code is used in chunk 21.

4.2 random number generators

```
6 <rngs 6> ≡      /*  
     define a random seed object */  
     std::random_device randomseed;      /*  
     Standard Mersenne twister random number engine */  
     std::mt19937_64 rng(randomseed());  
     gsl_rng * rngtype;  
 static void setup_rng(unsigned long int s)  
{  
     const gsl_rng_type*T;  
     gsl_rng_env_setup();  
     T = gsl_rng_default;  
     rngtype = gsl_rng_alloc(T);  
     gsl_rng_set(rngtype, s);  
 }
```

This code is used in chunk 21.

4.3 e^x with checks

compute e^x checking for over- and underflow

7

```
 $\langle e^x \rangle \equiv$ 
static double veldi(const double x, const double y)
{
    feclearexcept(FE_ALL_EXCEPT);
    const double d = pow(x, y);
    return (fetestexcept(FE_UNDERFLOW) ? 0. : (fetestexcept(FE_OVERFLOW) ? FLT_MAX : d));
}
```

This code is used in chunk 21.

4.4 $\lambda_{n;k_1,\dots,k_r;s}$

```

compute  $\lambda_{n;k_1,\dots,k_r;s}(1)$ 
8  $\langle \lambda_{n;k_1,\dots,k_r;s}(1) \rangle \equiv$ 
  static double lambdanks (const double n, const std::vector<unsigned int> &v_k ) {
    assert(v_k.size() > 0);
    assert ( std::all_of (v_k.cbegin(), v_k.cend(), [] (const auto k)
    {
      return k > 1;
    }
    ) );
    double d
    {}
    ;
    double k
    {}
    ;
    double f
    {1};
    const double r = static_cast<double>(v_k.size());
    std::unordered_map<unsigned int , unsigned int > counts
    {}
    ;
    /*  $(x)_m \equiv x(x-1)\cdots(x-m+1)$  */
    auto ff = [] (const double x, const unsigned int m)
    {
      return static_cast<double>(boost::math::falling_factorial(x, m));
    }
    ;
    for (std::size_t i = 0; i < v_k.size(); ++i) {
      f *= ff(static_cast<double>(v_k[i]) - 1. - ALPHA, v_k[i] - 1); /* count occurrence of each merger size */
      ++counts[v_k[i]];
      k += static_cast<double>(v_k[i]);
      d += lgamma(static_cast<double>(v_k[i] + 1));
    }
    assert(k < n + 1);
    const double s = n - k; const double p = static_cast<double> ( std::accumulate
      (counts.cbegin(), counts.cend(), 0, [] (double a, const auto &x)
    {
      return a + lgamma((double)x.second + 1);
    }
    ) );
    const double l = ((v_k.size() < 2 ? (v_k[0] < 3 ? 1. : 0) : 0) * CKAPPA) + (CEPS * veldi(ALPHA,
      r + s - 1) * tgamma(r + s) * f / tgamma(n)); /* § 4.3 */
    return (veldi(exp(1),
      (lgamma(n + 1.) - d) - lgamma(n - k + 1) - p) * l / (CKAPPA + (CEPS * (1 - ALPHA)))); }
```

This code is used in chunk 21.

4.5 generate group sizes

generate group sizes k_1, \dots, k_r summing to $myInt$

9 ⟨ sizes of groups 9 ⟩ ≡

```

static double GenPartitions (const unsigned int m, const unsigned int myInt, const
    unsigned int PartitionSize, unsigned int MinVal, unsigned int MaxVal,
    std::vector < std::pair < double , std::vector < unsigned int >>> &v_l_k, std::vector
    < double > &lrates_sorting ) { /*
        m is the given number of blocks; the partitions sum to myInt */
double lrate
{ }
;
double sumrates
{ }
; 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;
    const char a = (idx_Spill) ? ~((-3 >> (LeftRemain - MinVal)) << 2) : 11;
    const 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;
    assert(static_cast<unsigned int>(std::accumulate(partition.begin(), partition.end(), 0)) ≡ myInt);
    lrate = lambdaNs(static_cast<double>(m), partition);
    assert(lrate ≥ 0);
    v_l_k.push_back(std::make_pair(lrate, partition));
    rates_sorting.push_back(lrate);
    sumrates += lrate;
} while (idx_Dec ≤ idx_Last);
assert(sumrates ≥ 0);
return sumrates; }

```

This code is used in chunk 21.

4.6 group sizes up to a given number

get all partitions summing to a given number

10 $\langle \text{sizes up to number } 10 \rangle \equiv$

```
static double allmergers_sum_m (const unsigned int n,const unsigned
    int m, std::vector < std::pair < double , std::vector < unsigned
    int >>> &v_l_k, std::vector < double >&v_lrates_sort ) { /*
        n is number of blocks ; all partitions summing to m ≤ n */
    const std::vector < unsigned int > v_m
    {m}; /* § 4.4 */
    double sumr = lambdanks(static_cast<double>(n), v_m);
    v_l_k.push_back(std::make_pair(sumr, v_m));
    v_lrates_sort.push_back(sumr);
    if (m > 3) {
        for (unsigned int s = 2; s ≤ m/2; ++s) {
            assert(m > 2 * (s - 1)); /* § 4.5 */
            sumr += GenPartitions(n, m, s, 2, m - (2 * (s - 1)), v_l_k, v_lrates_sort);
        }
    }
    assert(sumr ≥ 0);
    return sumr; }
```

This code is used in chunk 21.

4.7 write partitions to files

```
write partitions to files gg_n_.txt
11 <partitions to files 11> ≡
static void ratesmergersfile (const unsigned int n, const std::vector < unsigned
    int > &v_indx, const std::vector < std::pair < double , std::vector <
    unsigned int >>> &vlk, const double s, std::vector < std::vector <
    double >> &a_cmf ) {
    assert(s > 0);
    double cmf
    {
    }
    ;
    std::ofstream f;
    f.open("gg_" + std::to_string(n) + ".txt", std::ios::app);
    a_cmf[n].clear();
    for (const auto &i:v_indx) { cmf += (vlk[i].first)/s;
        assert(cmf ≥ 0);
        a_cmf[n].push_back(cmf);
        assert((vlk[i].second).size() > 0);
        for (const auto &x:vlk[i].second)
        {
            f << x << ' ';
        }
        f << '\n';
    }
    f.close();
    assert(abs(cmf - 1.) < 0.999999); }
```

This code is used in chunk 21.

4.8 partitions when given number of blocks

```

12 < all partitions given number of blocks 12 > ≡
    static void allmergers_when_n_blocks (const unsigned int n, std::vector <
        double > &v__lambdan, std::vector < std::vector < std::vector < double >> &a__cmf ) {
        std::vector < std::pair < double , std::vector < unsigned int >>> vlk
    { }
    ; std::vector < double > ratetosort
    { }
    ;
    ratetosort.clear();
    double lambdan
    { }
    ;
    vlk.clear();
    assert(n > 1);
    for (unsigned int k = 2; k ≤ n; ++k) { /* the partition sums to k; the number of blocks is n; § 4.6 */
        lambdan += allmergers_sum_m(n, k, vlk, ratetosort);
    } /* record the total rate when n blocks; use for sampling time */
    assert(lambdan > 0);
    v__lambdan[n] = lambdan;
    std::vector < unsigned int > indx(ratetosort.size());
    std::iota(indx.begin(), indx.end(), 0);
    std::stable_sort (indx.begin(), indx.end(), [&ratetosort](const unsigned int x, const
        unsigned int y)
    {
        return ratetosort[x] > ratetosort[y];
    }
    ); /* § 4.7 */
    ratesmergersfile(n, indx, vlk, v__lambdan[n], a__cmf); }
```

This code is used in chunk 21.

4.9 all partitions

generate all partitions and rates

```
13 <partitions generate all 13> ≡  
  static void all_partitions_rates ( std::vector < double > &vlmn, std::vector < std::vector  
    < double >> &acmf )  
  {  
    for (unsigned int tmpn = 2; tmpn ≤ SAMPLE_SIZE; ++tmpn) { /*  
      § 4.8 */  
      allmergers_when_n_blocks(tmpn, vlmn, acmf);  
    }  
  }
```

This code is used in chunk 21.

4.10 read in partitions

read in partitions k_1, \dots, k_r from files $g_{\langle m \rangle}.txt$

14 \langle partitions read in 14 $\rangle \equiv$

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

This code is used in chunk 21.

4.11 split partitions

```
    return b > 1;
```

```
    }
```

```
) ; } }
```

This code is used in chunk 21.

4.12 split groups in partition

given a partition, return with each group split into boxes

```
1 void split_groups( const std::vector< unsigned int>& partition,
2 std::vector<unsigned int>& split_partition )
3 {
4     split_partition.clear();
5     for( const auto &k:partition){
6         split_blocks( k, split_partition ) ;
7     }
8
9     assert( std::accumulate(split_partition.cbegin(), split_partition.cend(),0) <=
10    std::accumulate(partition.cbegin(), partition.cend(),0));
11 }
```

16 ⟨separate entire partition 16⟩ ≡

```
void split_groups ( const std::vector < unsigned int > &partition, std::vector < unsigned
int > &split_partition ) {
    split_partition.clear();
    for (const auto &k:partition)
    {
        /*
         § 4.11 */
        split_blocks(k, split_partition);
    }
    assert(std::accumulate(split_partition.cbegin(), split_partition.cend(),
0) ≤ std::accumulate(partition.cbegin(), partition.cend(),0)); }
```

This code is used in chunk 21.

4.13 merge blocks

merge blocks and record continuing ones

```
1 static void update_tree( std::vector<unsigned int>& tree, const
2 std::vector<unsigned int>& mergers )
3 {
4     assert( mergers.size() > 0 );
5
6     assert( static_cast<std::size_t>(std::accumulate(mergers.cbegin(),
7 mergers.cend(), 0)) <= tree.size() ) ;
8     std::shuffle( tree.begin(), tree.end(), rng) ;
9     std::vector<unsigned int> newblocks(mergers.size());
10    std::size_t j {} ;
11
12    for( const auto &m: mergers ){
13        newblocks[j] = std::accumulate( std::crbegin(tree), std::crbegin(tree) + m, \
14        + 0);
15        tree.resize( tree.size() - m ) ;
16        ++j ;
17    }
18    tree.reserve( tree.size() + newblocks.size() );
19    tree.insert( tree.end(), newblocks.cbegin(), newblocks.cend() ) ;
20 }
```

17 ⟨update tree 17⟩ ≡

```
static void update_tree ( std::vector < unsigned int > &tree, const std::vector <
    unsigned int > &mergers ) {
    assert(mergers.size() > 0);
    assert(static_cast<std::size_t>(std::accumulate(mergers.cbegin(),mergers.cend(),
        0)) ≤ tree.size());
    std::shuffle(tree.begin(),tree.end(),rng);
    std::vector < unsigned int > newblocks(mergers.size());
    std::size_t j
    {}
    ;
    for (const auto &m:mergers)
    {
        newblocks[j] = std::accumulate(std::crbegin(tree),std::crbegin(tree) + m,0);
        tree.resize(tree.size() - m);
        ++j;
    }
    tree.reserve(tree.size() + newblocks.size());
    tree.insert(tree.end(),newblocks.cbegin(),newblocks.cend()); }
```

This code is used in chunk 21.

4.14 sample time and partitions until a merger occurs

```

1 static double until_merger(const std::size_t current_number_blocks, const
2 std::vector<double>& v_lambdan, const std::vector<double>& v_cmf,
3 std::vector<unsigned int>& v_merger_sizes, double &otimi)
4 {
5 std::vector<unsigned int> v_partition (SAMPLE_SIZE/2);
6 v_partition.reserve(SAMPLE_SIZE/2) ;
7
8 unsigned int lina {};
9 double t {};
10
11 auto newtime = [] (const double l, const double otimi) {
12     return (RHO > DBL_EPSILON ? log1p( -(RHO * exp(- RHO * otimi) / l) *
13 log(gsl_rng_uniform_pos(rngtype))/RHO : gsl_ran_exponential(rngtype, 1./l)) ;
14 };
15
16 auto samplemerger = [&v_cmf](void){
17     unsigned int j {};
18     const double u = gsl_rng_uniform( rngtype);
19     while( u > v_cmf[j]) { ++j; }
20
21     return j ;
22 } ;
23
24 v_merger_sizes.clear() ;
25 while( std::all_of(v_merger_sizes.cbegin(), v_merger_sizes.cend(), [] (const
26 auto m){ return m < 2;}))
27 {
28     t += newtime( v_lambdan[current_number_blocks], otimi);
29     otimi += t ;
30     lina = samplemerger();
31     readmergersizes( current_number_blocks, 1 + lina, v_partition) ;
32     split_groups( v_partition, v_merger_sizes) ;
33 }
34
35 assert( v_merger_sizes.size() > 0) ;
36 assert( std::all_of(v_merger_sizes.cbegin(), v_merger_sizes.cend(), [] (const
37 auto m){ return m > 1;}));
38
39 return t ;
40 }
41
42 {until merger 18} ==
43 static double until_merger ( const std ::size_t current_number_blocks, const
44 std::vector < double > &v_lambdan, const std::vector <
45 double > &v_cmf, std::vector < unsigned int > &v_merger_sizes,
46 double &otimi ) {
47     std::vector < unsigned int > v_partition(SAMPLE_SIZE/2);
48     v_partition.reserve(SAMPLE_SIZE/2);
49     unsigned int lina
50     {}
51 }
```

```

double t
{
;
    /*  

     * new time  $t = \log(1 - \log(1 - U)/\phi) / \rho$ ,  $\phi = \lambda_n \exp(\rho s) / \rho$ ,  $s$  is cumulative  

     * time */
auto newtime = [](const double l, const double otime)
{
    return (RHO > DBL_EPSILON ? log1p(-(RHO * exp(-RHO *  

        otime)/l) * log(gsl_rng_uniform_pos(rngtype)))/RHO :  

        gsl_ran_exponential(rngtype, 1./l));
}
;
auto samplemerger = [&v_cmf](void)
{
    unsigned int j
{
;
    const double u = gsl_rng_uniform(rngtype);
    while (u > v_cmf[j]) {
        ++j;
    }
    return j;
}
;
v_merger_sizes.clear(); while ( std::all_of (v_merger_sizes.cbegin(),  

    v_merger_sizes.cend(), [](const auto m)
{
    return m < 2;
}
))
{
    t += newtime(v_lambdan[current_number_blocks], otime);
    otime += t;
    lina = samplemerger();      /*  

     * § 4.10 */
    readmergersizes(current_number_blocks, 1 + lina, v_partition);      /*  

     * § 4.12 */
    split_groups(v_partition, v_merger_sizes);
}
assert(v_merger_sizes.size() > 0);
assert ( std::all_of (v_merger_sizes.cbegin(), v_merger_sizes.cend(),  

    [](const auto m)
{
    return m > 1;
}
));
return t; }

```

This code is used in chunk 21.

4.15 one experiment

```

1 static void one_experiment( std::vector<double>& v_l, std::vector<double>& \
+   ↪ v_r,
2 const std::vector<double>& v_lambda, const std::vector< std::vector< double \
+   ↪ >
3 >& a_cmf)
4 {

5 std::vector<unsigned int> tree (SAMPLE_SIZE, 1) ;
6 tree.reserve(SAMPLE_SIZE) ;
7 std::fill(v_l.begin(), v_l.end(), 0);

8 double t {};
9 double ot {} ;

10 std::vector<unsigned int> v_merger_sizes {};
11 v_merger_sizes.clear() ;
12 v_merger_sizes.reserve(2*SAMPLE_SIZE) ;
13 while( tree.size() > 1)
14 {

15 t = until_merger(tree.size(), v_lambda, a_cmf[tree.size()], \
+   ↪ v_merger_sizes,
16 ot);
17 update_lengths(tree, v_l, t);
18 update_tree(tree, v_merger_sizes);
19 }
20 assert( tree.back() == SAMPLE_SIZE);
21 update_ri( v_r, v_l);
22 }

19 ⟨a single experiment 19⟩ ≡
static void one_experiment ( std::vector < double > & v_l, std::vector < double > & v_r,
                           const std::vector < double > & v_lambda, const std::vector < std::vector
                           < double >> & a_cmf ) {
    std::vector < unsigned int > tree(SAMPLE_SIZE, 1);
    tree.reserve(SAMPLE_SIZE);
    std::fill(v_l.begin(), v_l.end(), 0);
    double t
    {}
    ;
    double ot
    {}
    ;
    std::vector < unsigned int > v_merger_sizes
    {}
    ;
    v_merger_sizes.clear();
    v_merger_sizes.reserve(2 * SAMPLE_SIZE);
    auto update_lengths = [&tree, &v_l](const double t) { for (const auto &b:tree)
    {
        v_l[0] += t;
    }
}

```

```

    v_l[b] += t;
}
};

while (tree.size() > 1) { /* § 4.14 */
    t = until_merger(tree.size(), v_lambdan, a_cmf[tree.size()],
                      v_merger_sizes, ot);
    update_lengths(t);
    update_tree(tree, v_merger_sizes);
}

assert(tree.back() == SAMPLE_SIZE);
const double d = v_l[0];
std::transform (v_l.cbegin(), v_l.cend(), v_r.begin(), v_r.begin(), [&d](const
auto &x, const auto &y)
{
    return y + (x/d);
});
}

```

This code is used in chunk 21.

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

```

approximate  $\mathbb{E}[R_i(n)]$ 
1 static void approximate_eri()
2 {
3 std::vector<double> vri (SAMPLE_SIZE) ;
4 vri.reserve(SAMPLE_SIZE) ;

5 std::vector<double> v__l (SAMPLE_SIZE);
6 v__l.reserve(SAMPLE_SIZE) ;

7 std::vector<double> v__lambdan (SAMPLE_SIZE + 1) ;
8 v__lambdan.reserve(SAMPLE_SIZE + 1) ;

9 std::vector< std::vector< double >> a__cmfs (SAMPLE_SIZE + 1,
10 std::vector<double> {} ) ;

11 all_partitions_rates(v__lambdan, a__cmfs );

12 int r = EXPERIMENTS + 1 ;

13 while( --r > 0)
14 {
15 one_experiment(v__l, vri, v__lambdan, a__cmfs);
16 }

17 for( std::size_t i = 1; i < SAMPLE_SIZE; ++i){
18 std::cout << (log( vri[i]/static_cast<double>(EXPERIMENTS)) - log1p(-
19 vri[i]/static_cast<double>(EXPERIMENTS))) << '\n' ;
20 }

20 ⟨go ahead – approximate  $\mathbb{E}[R_i(n)]$  20⟩ ≡
static void approximate_eri() {
    std::vector < double > vri(SAMPLE_SIZE);
    vri.reserve(SAMPLE_SIZE); std::vector < double > v__l(SAMPLE_SIZE);
    v__l.reserve(SAMPLE_SIZE); std::vector < double > v__lambdan(SAMPLE_SIZE + 1);
    v__lambdan.reserve(SAMPLE_SIZE + 1); std::vector < std::vector < double >>
        a__cmfs (SAMPLE_SIZE + 1, std::vector < double > {} ) ; /*  

§ 4.9 */
    all_partitions_rates(v__lambdan, a__cmfs);
    int r = EXPERIMENTS + 1;
    while ( --r > 0) { /*  

        § 4.15 */
        one_experiment(v__l, vri, v__lambdan, a__cmfs);
    }
    for (std::size_t i = 1; i < SAMPLE_SIZE; ++i) {
        std::cout << (log(vri[i]/static_cast<double>(EXPERIMENTS)) -
            log1p(-vri[i]/static_cast<double>(EXPERIMENTS))) << '\n';
    }
}

```

This code is used in chunk 21.

4.17 main

the *main* module

```

21      /*
       § 4.1 */
⟨ includes 5⟩      /*
       § 4.2 */
⟨ rngs 6⟩      /*
       § 4.3 */
⟨ ex 7⟩      /*
       § 4.4 */
⟨ λn;k1,...,kr;s(1) 8⟩      /*
       § 4.5 */
⟨ sizes of groups 9⟩      /*
       § 4.6 */
⟨ sizes up to number 10⟩      /*
       § 4.7 */
⟨ partitions to files 11⟩      /*
       § 4.8 */
⟨ all partitions given number of blocks 12⟩      /*
       § 4.9 */
⟨ partitions generate all 13⟩      /*
       § 4.10 */
⟨ partitions read in 14⟩      /*
       § 4.11 */
⟨ split partition into boxes 15⟩      /*
       § 4.12 */
⟨ separate entire partition 16⟩      /*
       § 4.13 */
⟨ update tree 17⟩      /*
       § 4.14 */
⟨ until merger 18⟩      /*
       § 4.15 */
⟨ a single experiment 19⟩      /*
       § 4.16 */
⟨ go ahead – approximate E[Ri(n)] 20⟩
int main(int argc, const char *argv[])
{
    /*
       § 4.2 */
    setup_rng(static_cast(std::size_t)(atoi(argv[1])));
    /*
       § 4.16 */
    approximate_eri();
    gsl_rng_free(rngtype);
    return 0;
}

```

5 conclusions and bibliography

We approximate $\mathbb{E}[R_i(n)]$ when the coalescent is the time-changed $\Omega\text{-}\delta_0$ -Poisson-Dirichlet($\alpha, 0$)-coalescent with time-change function $G(t) = \int_0^t e^{\rho s} ds$ for a given fixed $\rho \geq 0$. Figure 1 example approximation of $\mathbb{E}[R_i(n)]$

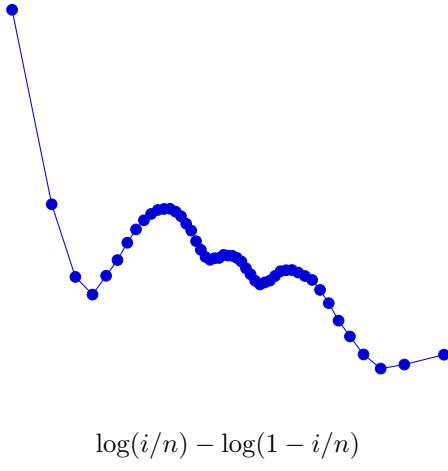


Figure 1: An example approximation of $\mathbb{E}[R_i(n)]$ for the given parameter values and graphed as logits against $\log(i/n) - \log(1 - i/n)$ for $i = 1, 2, \dots, n - 1$ where n is sample size

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/iya030>.
- [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

a: 8, 9.
a_cmf: 11, 12, 19.
a_cmfs: 20.
abs: 11.
accumulate: 8, 9, 15, 16, 17.
acmf: 13.
all_of: 8, 14, 18.
all_partitions_rates: 13, 20.
allmergers_sum_m: 10, 12.
allmergers_when_n_blocks: 12, 13.
ALPHA: 8.
app: 11.
approximate_eri: 20, 21.
argc: 21.
argv: 21.
assert: 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19.
atoi: 21.
b: 9, 15, 19.
back: 19.
back_inserter: 15.
begin: 9, 12, 17, 19.
boost: 8.
boxes: 15.
cbegin: 8, 14, 15, 16, 17, 18, 19.
cend: 8, 14, 15, 16, 17, 18, 19.
CEPS: 8.
CKAPPA: 8.
clear: 11, 12, 14, 16, 18, 19.
close: 11, 14.
cmf: 11.
copy_if: 15.
count_if: 15.
counts: 8.
cout: 20.
crbegin: 17.
current_number_blocks: 18.
d: 7, 8, 19.
DBL_EPSILON: 18.
double: 8, 18.
end: 9, 12, 17, 19.
exp: 8, 18.
EXPERIMENTS: 20.
f: 8.
falling_factorial: 8.
FE_ALL_EXCEPT: 7.
FE_OVERFLOW: 7.
FE_UNDERFLOW: 7.
feclearexcept: 7.
fetestexcept: 7.
ff: 8.
fill: 19.
first: 11.

FLT_MAX: 7.
GenPartitions: 9, 10.
getline: 14.
gsl_ran_exponential: 18.
gsl_rng: 6.
gsl_rng_alloc: 6.
gsl_rng_default: 6.
gsl_rng_env_setup: 6.
gsl_rng_free: 21.
gsl_rng_set: 6.
gsl_rng_type: 6.
gsl_rng_uniform: 15, 18.
gsl_rng_uniform_pos: 18.
i: 8, 11, 14, 20.
idx_Dec: 9.
idx_Last: 9.
idx_Spill: 9.
idx_SpillPrev: 9.
ifstream: 14.
indx: 12.
insert: 17.
ios: 11.
iota: 12.
istream_iterator: 14.
j: 14, 15, 17, 18.
k: 8, 12, 15, 16.
l: 8, 18.
lambdan: 12.
lambdanks: 8, 9, 10.
LeftRemain: 9.
lgamma: 8.
lina: 18.
log: 18, 20.
log1p: 18, 20.
lrate: 9.
lrates_sorting: 9.
m: 8, 9, 10, 17, 18.
main: 21.
make_pair: 9, 10.
math: 8.
MaxVal: 9.
mergers: 17.
MinVal: 9.
mt19937_64: 6.
myInt: 9.
n: 8, 10, 11, 12, 14.
newblocks: 17.
newtime: 18.
ofstream: 11.
one_experiment: 19, 20.
open: 11.
ot: 19.

`otime`: 18.
`otimi`: 18.
`p`: 8.
`pair`: 9, 10, 11, 12.
`partition`: 9, 16.
`PartitionSize`: 9.
`pow`: 7.
`push_back`: 9, 10, 11.
`r`: 8, 20.
`random_device`: 6.
`randomseed`: 6.
`ratesmergersfile`: 11, 12.
`ratetosort`: 12.
`readmergersizes`: 14, 18.
`reserve`: 17, 18, 19, 20.
`resize`: 17.
`RHO`: 18.
`rng`: 6, 17.
`rngtype`: 6, 15, 18, 21.
`s`: 6, 8, 10, 11.
`sample_box`: 15.
`SAMPLE_SIZE`: 13, 18, 19, 20.
`samplemerger`: 18.
`second`: 8, 11.
`setup_rng`: 6, 21.
`shuffle`: 17.
`size`: 8, 11, 12, 14, 17, 18, 19.
`split_blocks`: 15, 16.
`split_groups`: 16, 18.
`split_partition`: 15, 16.
`ss`: 14.
`stable_sort`: 12.
`std`: 6, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21.
`string`: 14.
`stringstream`: 14.
`sumr`: 10.
`sumrates`: 9.
`T`: 6.
`t`: 15, 18, 19.
`tgamma`: 8.
`tmpn`: 13.
`to_string`: 11, 14.
`transform`: 19.
`tree`: 17, 19.
`u`: 15, 18.
`unordered_map`: 8.
`until_merger`: 18, 19.
`update_lengths`: 19.
`update_tree`: 17, 19.
`v_cmf`: 18.
`v_idx`: 11.
`v_l`: 20.
`v_l_k`: 10.
`v_lambdan`: 12, 19, 20.
`v_m`: 10.
`v_merger_sizes`: 19.
`v_mergers`: 14.
`v_k`: 8.
`v_l`: 19.
`v_l_k`: 9.
`v_lambdan`: 18.
`v_lrates_sort`: 10.
`v_merger_sizes`: 18.
`v_partition`: 18.
`v_r`: 19.
`val_Dec`: 9.
`vector`: 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20.
`veldi`: 7, 8.
`vlk`: 11, 12.
`vlmn`: 13.
`void`: 15.
`vri`: 20.
`x`: 7, 8, 11, 12, 14, 19.
`y`: 7, 12, 19.

List of Refinements

$\langle \lambda_{n;k_1,\dots,k_r;s} (1) 8 \rangle$ Used in chunk 21.
 $\langle e^x 7 \rangle$ Used in chunk 21.
 \langle a single experiment 19 \rangle Used in chunk 21.
 \langle all partitions given number of blocks 12 \rangle Used in chunk 21.
 \langle go ahead – approximate $\mathbb{E}[R_i(n)]$ 20 \rangle Used in chunk 21.
 \langle includes 5 \rangle Used in chunk 21.
 \langle partitions generate all 13 \rangle Used in chunk 21.
 \langle partitions read in 14 \rangle Used in chunk 21.
 \langle partitions to files 11 \rangle Used in chunk 21.
 \langle rngs 6 \rangle Used in chunk 21.
 \langle separate entire partition 16 \rangle Used in chunk 21.
 \langle sizes of groups 9 \rangle Used in chunk 21.
 \langle sizes up to number 10 \rangle Used in chunk 21.
 \langle split partition into boxes 15 \rangle Used in chunk 21.
 \langle until merger 18 \rangle Used in chunk 21.
 \langle update tree 17 \rangle Used in chunk 21.