

# Gene genealogies in a haploid panmictic population evolving according to sweepstakes reproduction – sampling the Beta( $2 - \beta, \beta$ )-Poisson-Dirichlet( $\alpha, 0$ )-coalescent

BJARKI ELDON<sup>1,2</sup> 

Let  $\{\xi^n\} \equiv \{\xi^n(t) : t \geq 0\}$  be the Beta( $2 - \beta, \beta$ )-Poisson-Dirichlet( $\alpha, 0$ )-coalescent. Write  $\#A$  for the number of elements in a given finite set  $A$ ,  $L_i(n) \equiv \int_0^{\tau(n)} \#\{\xi \in \xi^n(t) : \#\xi = i\} dt$  and  $L(n) \equiv \int_0^{\tau(n)} \#\xi^n(t) dt$  and  $\tau(n) \equiv \inf\{t \geq 0 : \#\xi^n(t) = 1\}$  for  $i \in \{1, 2, \dots, n - 1\}$ . We then have  $L(n) = L_1(n) + \dots + L_{n-1}(n)$ . Define  $R_i(n) \equiv L_i(n)/\sum_j L_j(n)$  for  $i = 1, 2, \dots, n - 1$ . Interpreting  $\{\xi^n\}$  as ‘trees’ we may view  $L_i(n)$  as the random total length of branches supporting  $i \in \{1, 2, \dots, n - 1\}$  leaves, with the length measured in coalescent time units, and  $n$  sample size. With this C++ code we sample the Beta( $2 - \beta, \beta$ )-Poisson-Dirichlet( $\alpha, 0$ )-coalescent with  $0 < \alpha < 1$  and  $1 < \beta < 2$  fixed, and approximate  $\mathbb{E}[R_i(n)]$ .

## Contents

1	Copyright	2
2	compilation and output	3
3	intro	4
4	code	5
4.1	the includes . . . . .	6
4.2	random number generators . . . . .	7
4.3	descending factorial . . . . .	8
4.4	veldi . . . . .	9
4.5	the beta part in $\lambda_{n;k_1,\dots,k_r;s}$ . . . . .	10
4.6	$\lambda_{n;k_1,\dots,k_r;s}$ . . . . .	11
4.7	merger sizes . . . . .	12
4.8	all mergers with fixed sum . . . . .	14
4.9	record merger sizes in order . . . . .	15
4.10	all mergers when $n$ blocks . . . . .	16
4.11	all mergers up to sample size . . . . .	17
4.12	update tree . . . . .	18
4.13	update lengths . . . . .	19
4.14	update approximations $r_i$ . . . . .	20
4.15	sample merger sizes . . . . .	21
4.16	read merger sizes . . . . .	22
4.17	one experiment . . . . .	23
4.18	approximate . . . . .	24
4.19	main . . . . .	25

<sup>1</sup>beldon11@gmail.com

<sup>2</sup>compiled @ 3:56pm on Wednesday 15<sup>th</sup> October, 2025

CTANGLE 4.12.1 (TeX Live 2025/Debian)

g++ (Debian 15.2.0-4) 15.2.0

kernel 6.16.11+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

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  $\geq 3$ ). 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

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 [2]:  
parallel --gnu -j1 :::: /path/to/scriptfile
```

### 3 intro

Write  $\mathbb{N} \equiv \{1, 2, \dots\}$  and  $\mathbb{N}_0 \equiv \{0, 1, 2, \dots\}$  and  $[n] \equiv \{1, 2, \dots, n\}$  for all  $n \in \mathbb{N}$ . Fix  $0 < \alpha < 1$  and  $1 < \beta < 2$ . The Beta( $2 - \beta, \beta$ )-Poisson-Dirichlet( $\alpha, 0$ )-coalescent coalescent is a Markov-chain on the partitions on  $[n]$  with transition rates

$$C_\beta = \frac{\beta}{m_\infty^\beta} B(2 - \beta, \beta) \quad (1a)$$

$$\lambda_{n;k_1, \dots, k_r;s} = \mathbf{1}_{\{r=1\}} \frac{C_\beta}{C_\beta + c(1 - \alpha)} B(k - \beta, n - k + \beta) + \frac{c p_{n;k_1, \dots, k_r;s}}{C_\beta + c(1 - \alpha)} \quad (1b)$$

where the transition is the merging of blocks in  $r$  groups of size(s)  $k_1, \dots, k_r$ . We approximate  $m_\infty$  in (1a) with  $(2 + (1 + 2^{1-\beta}) / (\beta - 1)) / 2$ . In (1b)  $p_{n;k_1, \dots, k_r;s}$  is given by

$$p_{n;k_1, \dots, k_r;s} = \frac{\alpha^{r+s-1} (r+s-1)!}{(n-1)!} \prod_{i=1}^r (k_i - 1 - \alpha)_{k_i-1} \quad (2)$$

where  $(x)_m \equiv x(x-1) \cdots (x-m+1)$  and  $(x)_0 \equiv 1$  for  $x$  real and  $m \in \mathbb{N}_0$ .

The background for the Beta( $2 - \beta, \beta$ )-Poisson-Dirichlet( $\alpha, 0$ )-coalescent is this [1]. Fix  $0 < \alpha < 1$  and  $1 < \beta < 2$ . Consider a haploid population of constant finite size  $N$  evolving in non-overlapping generations. In each generation the current individuals independently produce potential offspring according to

$$\mathbb{P}(X = k) = C \left( k^{-a} - (k+1)^{-a} \right) \quad (3)$$

for all  $k = 1, 2, \dots, \zeta(N)$  where  $a > 0$  and  $C$  is such that  $\mathbb{P}(1 \leq X \leq \zeta(N)) = 1$ , and such that  $\zeta(N)/N^{1/\alpha} \rightarrow \infty$  as  $N \rightarrow \infty$ . Let  $X_1, \dots, X_N$  be the iid random numbers of potential offspring produced by the current  $N$  individuals. Write  $X_1, \dots, X_N \triangleright \mathbb{L}(a, \zeta(N))$  when have law (3). Take  $(\varepsilon_N)_N$  a positive sequence where  $0 < \varepsilon_N < 1$ . It may hold that  $\varepsilon_N \rightarrow 0$  as  $N \rightarrow \infty$ . With probability  $\varepsilon_N$  it holds that  $a = \alpha$ , and with probability  $1 - \varepsilon_N$  it holds that  $a = \beta$ , where  $a$  is the one in (3),

$$X_1, \dots, X_N \triangleright \mathbb{L}(\mathbf{1}_{\{E\}}\alpha + \mathbf{1}_{\{E^c\}}\beta, \zeta(N))$$

where  $E$  is the event that  $a = \alpha$ . Taking  $\varepsilon_N \stackrel{c}{\sim} c_N$  where  $N^{\beta-1}c_N \stackrel{c}{\sim} 1$  as  $N \rightarrow \infty$  leads to the coalescent with transition rates (1b).

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

## 4 code

we briefly summarise the algorithm, let  $\lambda_m$  denote the total rate of merging  $m$  blocks obtained by summing the rates (1b) over all possible ordered merger sizes given  $m$  blocks

1.  $r_i \leftarrow 0$  for  $i = 1, 2, \dots, n - 1$
2. for each of  $M$  experiments: § 4.18
  - (a)  $m \leftarrow n$
  - (b)  $\ell_i \leftarrow 0$  for all  $i \in [n - 1]$
  - (c) the current block sizes  $b_i \leftarrow 1$  for all  $i \in [n]$
  - (d) **while**  $m > 1$  : § 4.17
    - i. sample time until next merger  $t \leftarrow \text{Exp}(\lambda_m)$
    - ii. update  $\ell_b \leftarrow \ell_b + t$  for all  $b \in \{b_1, \dots, b_m\}$  § 4.13
    - iii. sample merger size(s) § 4.15  $k_1, \dots, k_r$  and merge blocks § 4.12
    - iv.  $m \leftarrow m - k_1 - \dots - k_r + r$
  - (e)  $r_i \leftarrow \ell_i / \sum_j \ell_j$  for all  $i \in [n - 1]$  § 4.14
3. return  $\bar{\varrho}_i(n)$  as  $r_i/M$  for all  $i \in [n - 1]$

#### 4.1 the includes

the included libraries

```
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 <gsl/gsl_sf.h>
#include "beta_poisson_dirichlet_usingrates.hpp"
```

This code is used in chunk 23.

## 4.2 random number generators

defining the random number generators

```
1 gsl_rng * rngtype ;
2 static void setup_rng(unsigned long int s)
3 {
4 const gsl_rng_type *T ;
5 gsl_rng_env_setup();
6 T = gsl_rng_default ;
7 rngtype = gsl_rng_alloc(T);
8 gsl_rng_set( rngtype, s) ;
9 }
```

6  $\langle \text{rngs } 6 \rangle \equiv$   
    *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 23.

### 4.3 descending factorial

the descending factorial  $(x)_m$

```
1 static double descending_factorial(const double x, const double m)
2 {
3     double p = 1;
4
5     for( double i = 0; i < m; ++i){
6         p *= (x - i); }
7
8 }descending 7} ≡
static double descending_factorial(const double x,const double m)
{
    double p = 1;
    for (double i = 0; i < m; ++i) {
        p *= (x - i);
    }
    return p;
}
```

This code is used in chunk 23.

#### 4.4 veldi

```
compute  $x^y$  checking for underflow and overflow
1 static double veldi (const double x, const double y)
2 {
3     feclearexcept (FE_ALL_EXCEPT);
4     const double d = pow (x,y);
5
6     return (fetestexcept (FE_UNDERFLOW) ? 0. : (fetestexcept (FE_OVERFLOW) ?
7         FLT_MAX : d));
}
8 {veldi 8} ≡
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 23.

#### 4.5 the beta part in $\lambda_{n;k_1,\dots,k_r;s}$

compute the beta part in  $\lambda_{n;k_1,\dots,k_r;s}$  (1b), the

$$\frac{C_\beta}{C_\beta + c(1 - \alpha)} B(k - \beta, n - k + \beta)$$

part

```

1 double betapart (const double n, const double k)
2 {
3     return (CBETA * gsl_sf_beta (k - BETA, n - k + BETA)) / (CBETA +
4     CEPS*(1-ALPHA)) ;
5 }
```

9  $\langle \text{beta part 9} \rangle \equiv$

```

double betapart(const double n,const double k)
{
    /*
        CBETA is  $C_\beta$ ; BETA is  $\beta$ ; ALPHA is  $\alpha$  */
    return (CBETA * gsl_sf_beta(k - BETA, n - k + BETA)) / (CBETA + (CEPS * (1 - ALPHA)));
}
```

This code is used in chunk 23.

#### 4.6 $\lambda_{n;k_1,\dots,k_r;s}$

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

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

```

10  < lambdanks 10 > ≡
    static double lambdanks (const double n, const std::vector<unsigned int> &v_k ) {
        assert(v_k[0] > 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
        {};
        ;
        for (std::size_t i = 0; i < v_k.size(); ++i) { /* § 4.3 */
            f *= descending_factorial(static_cast<double>(v_k[i]) - 1. - ALPHA,
                static_cast<double>(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.begin(), counts.end(), 0,
            [] (double a, const auto &x)
        {
            return a + lgamma((double)x.second + 1);
        }
        ) ); /* betapart § 4.5 */
        const double l = ((v_k.size() < 2 ? 1. : 0) * betapart(n, v_k[0])) + (CEPS * veldi(ALPHA,
            r + s - 1) * tgamma(r + s) * f / tgamma(n));
        return (veldi(exp(1),
            (lgamma(n + 1.) - d) - lgamma(n - k + 1) - p) * l / (CBETA + (CEPS * (1 - ALPHA)))); }
```

This code is used in chunk 23.

## 4.7 merger sizes

generate merger size(s) of  $m$  groups summing to  $myInt$

11  $\langle$  merger sizes 11  $\rangle \equiv$

```

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);
    /* § 4.6 */
    lrate = lambdanks(static_cast(double)(m), partition);
    assert(lrate ≥ 0);
    v_l_k.push_back(std::make_pair(lrate, partition));
    lrates_sorting.push_back(lrate);
    sumrates += lrate;
} while (idx_Dec ≤ idx_Last);
assert(sumrates ≥ 0);
return sumrates; }

```

This code is used in chunk 23.

#### 4.8 all mergers with fixed sum

get all mergers summing to  $m$

12  $\langle \text{mergers fixed sum } 12 \rangle \equiv$

```

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 the number of blocks; the partitions sum to m */
const std::vector < unsigned int > v_m
{m}; /*

    § 4.6; first record  $\lambda_{n;m;n-m}$  the rate of a single merger of size m */

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)); /*

            § 4.7; add the rates of simultaneous mergers */

        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 23.

#### 4.9 record merger sizes in order

record ordered merger sizes

13 ⟨ record merger sizes in order 13 ⟩ ≡

```
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 23.

#### 4.10 all mergers when $n$ blocks

get all mergers when a given number of blocks

14  $\langle$  mergers when  $n$  blocks 14  $\rangle \equiv$

```

static void allmergers_when_n_blocks (const unsigned int n, std::vector <
    double > &v_lambdan, 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; § 4.8 */
    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 > idx(ratetosort.size());
std::iota(idx.begin(), idx.end(), 0); std::stable_sort (idx.begin(), idx.end(),
    [&ratetosort](const unsigned int x, const unsigned int y)
{
    return ratetosort[x] > ratetosort[y];
}
); /* merger rates sorted in descending order; print the cmf and rates to file; § 4.9 */
ratesmergersfile(n, idx, vlk, v_lambdan[n], a_cmf); }
```

This code is used in chunk 23.

#### 4.11 all mergers up to sample size

generate all mergers up to sample size

```
15 <all mergers 15> ≡  
  static void allmergers ( std::vector < double > &vlmn, std::vector < std::vector <  
    double >> &acmf )  
  {  
    for (unsigned int tmpn = 2; tmpn ≤ SAMPLE_SIZE; ++tmpn) { /*  
      § 4.10 */  
      allmergers_when_n_blocks(tmpn, vlmn, acmf);  
    }  
  }
```

This code is used in chunk 23.

#### 4.12 update tree

```
update block sizes
16 < update block sizes 16 > ≡
    static void updatetree ( std::vector < unsigned int > &tre, const std::vector < unsigned
        int > &mergersizes ) {
        assert(mergersizes.size() > 0);
        std::vector < unsigned int > newblocks
        {}
        ;
        newblocks.clear();
        std::shuffle(tre.begin(), tre.end(), rng);
        std::size_t s = tre.size(); for (const auto &k:mergersizes)
        {
            assert(k > 1);
            assert(k ≤ s);
            s -= k; /* record the size of the merging blocks */
            newblocks.push_back(std::accumulate(tre.rbegin(), tre.rbegin() + k, 0)); /* remove the blocks that merged */
            tre.resize(s);
        }
        assert(newblocks.size() > 0);
        assert(static_cast<unsigned int>(std::accumulate(newblocks.begin(), newblocks.end(),
            0)) ≤ SAMPLE_SIZE);
        tre.insert(tre.end(), newblocks.begin(), newblocks.end());
        assert(static_cast<unsigned int>(std::accumulate(tre.begin(), tre.end(),
            0)) ≡ SAMPLE_SIZE); }
```

This code is used in chunk 23.

#### 4.13 update lengths

update the lengths  $\ell_i$  given the current block sizes

17  $\langle \text{lengths } 17 \rangle \equiv$

```
static void updatelengths ( const std::vector < unsigned int > &tre, std::vector <
    double > &v_lengths, const std::vector < double > &v_lambdan ) { /* get the time until merger */
    const double t = gsl_ran_exponential(rngtype, 1./v_lambdan[tre.size()]);
    for (const auto &b:tre)
    {
        assert(b > 0);
        assert(b < SAMPLE_SIZE);
        v_lengths[0] += t;
        v_lengths[b] += t;
    }
}
```

This code is used in chunk 23.

#### 4.14 update approximations $r_i$

update the approximations  $r_i$

18  $\langle r_i \text{ 18} \rangle \equiv$

```
static void updateri ( const std::vector < double > &v__l, std::vector < double > &v_ri
) {
assert(v__l[0] > 0);
const double d = v__l[0]; std::transform (v__l.begin(), v__l.end(), v_ri.begin(),
v_ri.begin(), [&d](const auto &x, const auto &y)
{
    return y + (x/d);
}
); }
```

This code is used in chunk 23.

#### 4.15 sample merger sizes

```
get merger size(s)
19 <get merger sizes 19> ≡
    static unsigned int samplemerger (const unsigned int n, const std::vector<
        double > &v__cmf )
    {
        unsigned int j
        {}
        ;
        const double u = gsl_rng_uniform(rngtype);
        while (u > v__cmf[j]) {
            ++j;
        } /* j is the sampled index of the ordered merger size(s) */
        return j;
    }
```

This code is used in chunk 23.

#### 4.16 read merger sizes

```
20 < read in merger sizes 20 >≡
  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(v__mergers[0] > 1);
    assert(v__mergers.back() > 1);
    f.close(); }
```

This code is used in chunk 23.

#### 4.17 one experiment

generate one experiment

```
21 <one experiment 21> ≡
    static void onexperiment ( std::vector < double > &v__ri, std::vector < double > &vl,
        const std::vector < double > &v__lambda__n, const std::vector <
        std::vector < double >> &a__cmf ) {
        std::vector < unsigned int > v__tre(SAMPLE_SIZE, 1);
        std::fill(vl.begin(), vl.end(), 0);
        unsigned int lina
        {
        ;
        std::vector < unsigned int > v__merger_sizes(SAMPLE_SIZE/2);
        v__merger_sizes.reserve(SAMPLE_SIZE/2);
        while (v__tre.size() > 1) { /* § 4.13 */
            update lengths(v__tre, vl, v__lambda__n); /* § 4.15 */
            lina = samplemerger(v__tre.size(), a__cmf[v__tre.size()]); /* § 4.16 */
            readmergersizes(v__tre.size(), 1 + lina, v__merger_sizes); /* § 4.12 */
            updatetree(v__tre, v__merger_sizes);
        }
        assert(v__tre.back() ≡ SAMPLE_SIZE); /* § 4.14 */
        updateri(vl, v__ri); }
```

This code is used in chunk 23.

#### 4.18 approximate

```
22 < go ahead - get  $\bar{\varrho}_i(n)$  22 > ≡
static void approximate() {
    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_l_n(SAMPLE_SIZE + 1);
    v_l_n.reserve(SAMPLE_SIZE + 1);
    std::vector < std::vector < double >> a_cmfs (SAMPLE_SIZE + 1, std::vector <
        double > { } ) ; /*
        § 4.11 */
    allmergers(v_l_n, a_cmfs);
    int r = EXPERIMENTS + 1;
    while (--r > 0) { /*
        § 4.17 */
        onexperiment(vri, v_l, v_l_n, a_cmfs);
    }
    for (const auto &z:vri)
    {
        std::cout ≪ z ≪ '\n';
    }
}
```

This code is used in chunk 23.

#### 4.19 main

the *main* function

```

23      /*
§ 4.1 */
⟨ includes 5⟩    /*
§ 4.2 */
⟨ rngs 6⟩    /*
§ 4.3 */
⟨ descending 7⟩    /*
§ 4.4 */
⟨ veldi 8⟩    /*
§ 4.5 */
⟨ beta part 9⟩    /*
§ 4.6 */
⟨ lambdanks 10⟩    /*
§ 4.7 */
⟨ merger sizes 11⟩    /*
§ 4.8 */
⟨ mergers fixed sum 12⟩    /*
§ 4.9 */
⟨ record merger sizes in order 13⟩    /*
§ 4.10 */
⟨ mergers when n blocks 14⟩    /*
§ 4.11 */
⟨ all mergers 15⟩    /*
§ 4.12 */
⟨ update block sizes 16⟩    /*
§ 4.13 */
⟨ lengths 17⟩    /*
§ 4.14 */
⟨  $r_i$  18⟩    /*
§ 4.15 */
⟨ get merger sizes 19⟩    /*
§ 4.16 */
⟨ read in merger sizes 20⟩    /*
§ 4.17 */
⟨ one experiment 21⟩    /*
§ 4.18 */
⟨ go ahead – get  $\bar{\varrho}_i(n)$  22⟩

int main(int argc, char *argv[])
{
    /*
§ 4.2 */
    setup_rng(static_cast<unsigned long>(atoi(argv[1])));
    /*
§ 4.18 */
    approximate();
    gsl_rng_free(rngtype);
    return 0;
}

```

## 5 conclusions and references

we sample the Beta( $2 - \beta, \beta$ )-Poisson-Dirichlet( $\alpha, 0$ )-coalescent and approximate  $\mathbb{E}[R_i(n)]$ . An example approximation in Figure 1.

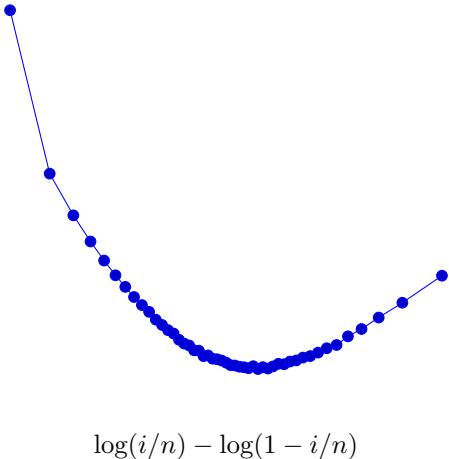


Figure 1: An example approximation of  $\mathbb{E}[R_i(n)]$  for the given parameter values and graphed as logits against  $\log(i/n) - \log(1 - i/n)$  for  $i = 1, 2, \dots, n - 1$  where  $n$  is sample size;

## References

- [1] ELDON, B. Gene genealogies in haploid populations evolving according to sweepstakes reproduction. In preparation, 2025+.
- [2] TANGE, O. GNU parallel – the command-line power tool. The USENIX Magazine, 2011.

# Index

*a:* 10, 11.  
*a\_cmf:* 13, 14, 21.  
*a\_cmfs:* 22.  
*abs:* 13.  
*accumulate:* 10, 11, 16.  
*acmf:* 15.  
*allmergers:* 15, 22.  
*allmergers\_sum\_m:* 12, 14.  
*allmergers\_when\_n\_blocks:* 14, 15.  
**ALPHA:** 9, 10.  
*app:* 13.  
*approximate:* 22, 23.  
*argc:* 23.  
*argv:* 23.  
*assert:* 10, 11, 12, 13, 14, 16, 17, 18, 20, 21.  
*atoi:* 23.  
*b:* 11, 17.  
*back:* 20, 21.  
*begin:* 10, 11, 14, 16, 18, 21.  
**BETA:** 9.  
*betapart:* 9, 10.  
**CBETA:** 9, 10.  
**CEPS:** 9, 10.  
*clear:* 13, 14, 16, 20.  
*close:* 13, 20.  
*cmf:* 13.  
*counts:* 10.  
*cout:* 22.  
*d:* 8, 10, 18.  
*descending\_factorial:* 7, 10.  
**double:** 10.  
*end:* 10, 11, 14, 16, 18, 21.  
*exp:* 10.  
**EXPERIMENTS:** 22.  
*f:* 10.  
**FE\_ALL\_EXCEPT:** 8.  
**FE\_OVERFLOW:** 8.  
**FE\_UNDERFLOW:** 8.  
*feclearexcept:* 8.  
*fetestexcept:* 8.  
*fill:* 21.  
*first:* 13.  
**FLT\_MAX:** 8.  
*GenPartitions:* 11, 12.  
*getline:* 20.  
*gsl\_ran\_exponential:* 17.  
*gsl\_rng:* 6.  
*gsl\_rng\_alloc:* 6.  
*gsl\_rng\_default:* 6.  
*gsl\_rng\_env\_setup:* 6.  
*gsl\_rng\_free:* 23.  
*gsl\_rng\_set:* 6.  
*gsl\_rng\_type:* 6.  
*gsl\_rng\_uniform:* 19.  
*gsl\_sf\_beta:* 9.  
*i:* 7, 10, 13, 20.  
*idx\_Dec:* 11.  
*idx\_Last:* 11.  
*idx\_Spill:* 11.  
*idx\_SpillPrev:* 11.  
*ifstream:* 20.  
*indx:* 14.  
*insert:* 16.  
*ios:* 13.  
*iota:* 14.  
*istream\_iterator:* 20.  
*j:* 19, 20.  
*k:* 9, 10, 14, 16.  
*l:* 10.  
*lambdan:* 14.  
*lambdanks:* 10, 11, 12.  
*LeftRemain:* 11.  
*lgamma:* 10.  
*lina:* 21.  
*lrate:* 11.  
*lrates\_sorting:* 11.  
*m:* 7, 11, 12.  
*main:* 23.  
*make\_pair:* 11, 12.  
*MaxVal:* 11.  
*mergersizes:* 16.  
*MinVal:* 11.  
*mt19937\_64:* 6.  
*myInt:* 11.  
*n:* 9, 10, 12, 13, 14, 19, 20.  
*newblocks:* 16.  
*ofstream:* 13.  
*onexperiment:* 21, 22.  
*open:* 13.  
*p:* 7, 10.  
*pair:* 11, 12, 13, 14.  
*partition:* 11.  
*PartitionSize:* 11.  
*pow:* 8.  
*push\_back:* 11, 12, 13, 16.  
*r:* 10, 22.  
*random\_device:* 6.  
*randomseed:* 6.  
*ratesmergersfile:* 13, 14.  
*ratetosort:* 14.  
*rbegin:* 16.  
*readmergersizes:* 20, 21.  
*reserve:* 21, 22.  
*resize:* 16.

<i>rng</i> :	6, 16.	<i>vri</i> :	22.
<i>rngtype</i> :	6, 17, 19, 23.	<i>x</i> :	7, 8, 10, 13, 14, 18.
<i>s</i> :	6, 10, 12, 13, 16.	<i>y</i> :	8, 14, 18.
<i>SAMPLE_SIZE</i> :	15, 16, 17, 21, 22.	<i>z</i> :	22.
<i>samplemerger</i> :	19, 21.		
<i>second</i> :	10, 13.		
<i>setup_rng</i> :	6, 23.		
<i>shuffle</i> :	16.		
<i>size</i> :	10, 13, 14, 16, 17, 20, 21.		
<i>ss</i> :	20.		
<i>stable_sort</i> :	14.		
<i>std</i> :	6, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22.		
<i>string</i> :	20.		
<i>stringstream</i> :	20.		
<i>sumr</i> :	12.		
<i>sumrates</i> :	11.		
<i>T</i> :	6.		
<i>t</i> :	17.		
<i>tgamma</i> :	10.		
<i>tmpn</i> :	15.		
<i>to_string</i> :	13, 20.		
<i>transform</i> :	18.		
<i>tre</i> :	16, 17.		
<i>u</i> :	19.		
<i>unordered_map</i> :	10.		
<i>update_lengths</i> :	17, 21.		
<i>updateri</i> :	18, 21.		
<i>updatetree</i> :	16, 21.		
<i>v_cmf</i> :	19.		
<i>v_idx</i> :	13.		
<i>v_l</i> :	18, 22.		
<i>v_l_k</i> :	12.		
<i>v_lambda_n</i> :	21.		
<i>v_lambda_n</i> :	14.		
<i>v_lengths</i> :	17.		
<i>v_m</i> :	12.		
<i>v_mergers</i> :	20.		
<i>v_ri</i> :	21.		
<i>v_tre</i> :	21.		
<i>v_k</i> :	10.		
<i>v_l_k</i> :	11.		
<i>v_l_n</i> :	22.		
<i>v_lambda_n</i> :	17.		
<i>v_lrates_sort</i> :	12.		
<i>v_merger_sizes</i> :	21.		
<i>v_ri</i> :	18.		
<i>val_Dec</i> :	11.		
<i>vector</i> :	10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22.		
<i>veldi</i> :	8, 10.		
<i>vl</i> :	21.		
<i>vlk</i> :	13, 14.		
<i>vlmn</i> :	15.		

## List of Refinements

$\langle r_i \rangle$  Used in chunk 23.  
 $\langle$  all mergers  $15 \rangle$  Used in chunk 23.  
 $\langle$  beta part  $9 \rangle$  Used in chunk 23.  
 $\langle$  descending  $7 \rangle$  Used in chunk 23.  
 $\langle$  get merger sizes  $19 \rangle$  Used in chunk 23.  
 $\langle$  go ahead – get  $\bar{\varrho}_i(n) 22 \rangle$  Used in chunk 23.  
 $\langle$  includes  $5 \rangle$  Used in chunk 23.  
 $\langle$  lambdanks  $10 \rangle$  Used in chunk 23.  
 $\langle$  lengths  $17 \rangle$  Used in chunk 23.  
 $\langle$  merger sizes  $11 \rangle$  Used in chunk 23.  
 $\langle$  mergers fixed sum  $12 \rangle$  Used in chunk 23.  
 $\langle$  mergers when  $n$  blocks  $14 \rangle$  Used in chunk 23.  
 $\langle$  one experiment  $21 \rangle$  Used in chunk 23.  
 $\langle$  read in merger sizes  $20 \rangle$  Used in chunk 23.  
 $\langle$  record merger sizes in order  $13 \rangle$  Used in chunk 23.  
 $\langle$  rngs  $6 \rangle$  Used in chunk 23.  
 $\langle$  update block sizes  $16 \rangle$  Used in chunk 23.  
 $\langle$  veldi  $8 \rangle$  Used in chunk 23.