Gene genealogies in diploid populations evolving according to sweepstakes reproduction — approximating $\mathbb{E}[R_i(n)]$ for the time-changed Ω-δ_0-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 $L(n) = L_1(n) + \cdots + L_{n-1}(n)$. Write $R_i(n) \equiv L_i(n)/L(n)$ for $i = 1, 2, \ldots, n-1$. 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-Poisson-Dirichlet$(\alpha, 0)$ coalescent with $0 < \alpha < 1$ and time-change function $G(t)$.

Contents

[1] beldon11@gmail.com

[2] compiled @ 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

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 $2 \leq k_1, \ldots, k_r \leq n$ with $\sum_i k_i \leq n$, $s = n - \sum_i k_i$, according to

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

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 $L(n) = L_1(n) + \cdots + L_{n-1}(n)$. Write $R_i(n) \equiv L_i(n)/L(n)$ for $i = 1, 2, \ldots, n-1$. 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 $\langle\, e^x\ 7\,\rangle \equiv$

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

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

8 $\langle \lambda_{n;k_1,\ldots,k_r;s}$ (1) 8 $\rangle \equiv$

```
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 ≡ x(x − 1) · · · (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 $k_1, \ldots, k_r$ 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 = 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 21.

4.6 group sizes up to a given number

get all partitions summing to a given number

10 ⟨sizes up to number 10⟩ ≡

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** $m \leq n$ */

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** $s = 2$; $s \leq m/2$; ++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 < 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 $k_1, \ldots, k_r$ from files g_<m>_.txt

14 ⟨partitions read in 14⟩ ≡

```
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

```
void split_blocks(const unsigned int k, std::vector<unsigned int>&
split_partition )
{
auto sample_box = [](void)
{ const double u = gsl_rng_uniform(rngtype);
return static_cast<unsigned int>(u < 0.25 ? 0 : ( u < .5 ? 1 : ( u < .75 ? 2 :
3))) ;
} ;

std::vector<unsigned int> boxes (4);
for( unsigned int j = 0; j < k; ++j)
{
++boxes[ sample_box()  ];
}
assert( static_cast<unsigned int>( std::accumulate(boxes.cbegin(),
boxes.cend(), 0))  == k);

const auto t = static_cast<std::size_t>( std::count_if( boxes.cbegin(),
boxes.cend(), [](const auto b){ return b > 1;}));

if(t > 0){
std::copy_if(boxes.cbegin(), boxes.cend(), \
std::back_inserter(split_partition),
[](const auto b){return b > 1;});
}
}
```

15 ⟨split partition into boxes 15⟩ ≡

```
void split_blocks (const unsigned int k, std::vector < unsigned int > &split_partition ) {
auto sample_box = [](void)
{
  const double u = gsl_rng_uniform(rngtype);
  return static_cast⟨unsigned int⟩(u < 0.25 ? 0 : (u < .5 ? 1 : (u < .75 ? 2 : 3)));
}
;
std::vector < unsigned int > boxes(4);
for (unsigned int j = 0; j < k; ++j) {
  ++boxes[sample_box()];
}
assert(static_cast⟨unsigned int⟩(std::accumulate(boxes.cbegin(), boxes.cend(), 0)) ≡ k);
  /*
    count how many boxes have at least 2 blocks */
const auto t = static_cast⟨std::size_t⟩ ( std::count_if (boxes.cbegin(), boxes.cend(),
        [](const auto b)
{
  return b > 1;
}
) ) ;
if (t > 0) { std::copy_if (boxes.cbegin(), boxes.cend(), std::back_inserter(split_partition),
        [](const auto b)
{
```

```
    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

```
void split_groups( const std::vector< unsigned int>& partition,
std::vector<unsigned int>& split_partition )
{
split_partition.clear();
for( const auto &k:partition){
split_blocks( k,  split_partition ) ;
}

assert( std::accumulate(split_partition.cbegin(), split_partition.cend(),0) <=
std::accumulate(partition.cbegin(), partition.cend(),0));
}
```

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) $\leq$ *std* :: *accumulate*(*partition.cbegin*( ), *partition.cend*( ), 0)); }

This code is used in chunk 21.

4.13 merge blocks

merge blocks and record continuing ones

```
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()  );
}
```

17 $\langle$update tree 17$\rangle \equiv$

static void *update_tree* (*std* :: *vector* < **unsigned int** > &*tree*, **const** *std* :: *vector* < **unsigned int** > &*mergers*) {

$assert(mergers.size() > 0);$

$assert(\textbf{static\_cast}\langle std::\textbf{size\_t}\rangle(std::accumulate(mergers.cbegin(), mergers.cend(), 0)) \leq tree.size());$

$std::shuffle(tree.begin(), tree.end(), rng);$

$std::vector < \textbf{unsigned int} > newblocks(mergers.size());$

$std::\textbf{size\_t}\ j$
{ }
;

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

```
static double until_merger(const std::size_t current_number_blocks, const
std::vector<double>& v_lambdan, const std::vector<double>& v__cmf,
std::vector<unsigned int>& v_merger_sizes, double &otimi)
{
std::vector<unsigned int> v_partition  (SAMPLE_SIZE/2);
v_partition.reserve(SAMPLE_SIZE/2) ;

unsigned int lina {} ;
double t {};

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], otimi);
otimi += t ;
lina = samplemerger();
readmergersizes( current_number_blocks, 1 + lina, v_partition) ;
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 ;
}
```

18 ⟨until merger 18⟩ ≡

static double *until_merger* (**const** *std* ::**size_t** *current_number_blocks*, **const** *std*::*vector* < **double** > &*v_lambdan*, **const** *std*::*vector* < **double** > &*v___cmf*, *std*::*vector* < **unsigned int** > &*v_merger_sizes*, **double** &*otimi*) {

std::*vector* < **unsigned int** > *v_partition*(SAMPLE_SIZE/2);

v_partition.*reserve*(SAMPLE_SIZE/2);

unsigned int *lina*

{ }

;

```
double t
{ }
;    /*
   new time t = log (1 − log(1 − U)/φ) /ρ, φ = λ_n exp(ρs)/ρ, 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], otimi);
  otimi += 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

```
static void one_experiment( std::vector<double>& v_l, std::vector<double>& \
↪ v_r,
const std::vector<double>& v__lambdan, 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) ;
while( tree.size() > 1)
{

t = until_merger(tree.size(), v__lambdan, a__cmf[tree.size()], \
↪ v__merger_sizes,
ot);
update_lengths(tree, v_l, t);
update_tree(tree, v__merger_sizes);
}
assert( tree.back() == SAMPLE_SIZE);
update_ri( v_r, v_l);
}
```

19 ⟨a single experiment 19⟩ ≡

static void *one_experiment* (*std*::*vector* < **double** > &*v_l*, *std*::*vector* < **double** > &*v_r*, **const** *std*::*vector* < **double** > &*v__lambdan*, **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 $\mathbb{E}[R_i(n)]$

```
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> {} ) ;

all_partitions_rates(v__lambdan, a__cmfs );

int r = EXPERIMENTS + 1 ;

while( --r > 0)
{
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'; }
}
```

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 */
⟨ e^x 7 ⟩ /* § 4.4 */
⟨ $\lambda_{n;k_1,\ldots,k_r;s}$ (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 $\mathbb{E}\left[R_i(n)\right]$ 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 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 $\rho \geq 0$. Figure 1 example approximation of $\mathbb{E}[R_i(n)]$

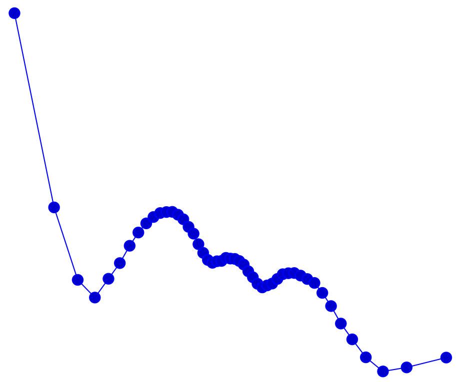

Figure 1: *An example approximation of $\mathbb{E}[R_i(n)]$ for the given parameter values and graphed as logits against $\log(i/n) - \log(1 - i/n)$ for $i = 1, 2, \ldots, n-1$ where n is sample size*

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/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

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

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

Index

List of Refinements