Gene genealogies in diploid populations evolving according to sweepstakes reproduction — approximating $\mathbb{E}[R_i(n)]$ for the 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 $\{\xi^n\} \equiv \{\xi^n(t) : t \geq 0\}$ is the Ω-δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent with $0 < \alpha < 1$, i.e. the δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent derived from a diploid panmictic population of constant size evolving according to a specific model of sweepstakes reproduction (heavy-tailed offspring number distribution) without selfing (each diploid offspring has 2 diploid parents).

Contents

[1] beldon11@gmail.com

[2] compiled @ 9:53am on Friday 9th January, 2026
CTANGLE 4.12.1 (TeX Live 2025/Debian)
g++ (Debian 15.2.0-12) 15.2.0
kernel 6.18.3+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 XeTeX 3.141592653-2.6-0.999997 (TeX Live 2025/Debian)
written using GNU Emacs 30.2

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

```
NAFN=diploid_delta_poisson_dirichlet
echo 'const unsigned int SAMPLE_SIZE = 50;' > $NAFN.hpp
echo 'const double ALPHA = 0.01;' >> $NAFN.hpp
echo 'const double KAPPA = 2;' >> $NAFN.hpp
echo 'const double CEPS = 100;' >> $NAFN.hpp
echo 'const double MINF = (2. + (1 + pow(2., 1-KAPPA)/(KAPPA -1)))/2. ;' >>
$NAFN.hpp
echo 'const double BKAPPA = pow(2,KAPPA)*(KAPPA - 2)*(KAPPA - 1);' >> $NAFN.hpp
echo 'const double AKAPPA = ((KAPPA+2) + pow(KAPPA,2))/2.;' >> $NAFN.hpp
echo 'const double CKAPPA = ((KAPPA > 2 ? AKAPPA / BKAPPA : 1.) *2.) /
pow(MINF,2.) ;' >> $NAFN.hpp
echo 'const int EXPERIMENTS = 1e5 ;' >> $NAFN.hpp
ctangle $NAFN.w
NAFN=diploid_delta_poisson_dirichlet
g++ -std=c++26 -m64 -march=native -O3 -x c++ $NAFN.c -lm -lgsl -lgslcblas
rm -f gg_*_.txt
./a.out $(shuf -i 323383-1919101019 -n1) P sed '1d' P awk '{S=1e5;print
log($1/S) - log(1 - ($1/S))}' > logitresout
seq 49 P awk '{S=50;print log($1/S) - log(1 - ($1/S))}' > nlogits
paste -d',' nlogits logitresout > forplottingfile1
sed -i 's/ALPHA = 0.01/ALPHA = 0.99/g' $NAFN.hpp
sed -i 's/CEPS = 100/CEPS = 1/g' $NAFN.hpp
ctangle $NAFN.w
g++ -std=c++26 -m64 -march=native -O3 -x c++ $NAFN.c -lm -lgsl -lgslcblas
rm -f gg_*_.txt
./a.out $(shuf -i 323383-1919101019 -n1) P sed '1d' P awk '{S=1e5;print
log($1/S) - log(1 - ($1/S))}' > logitresout
paste -d',' nlogits logitresout > forplottingfile2
cweave $NAFN.w
tail -n4 $NAFN.tex > endi
for i in $(seq 5); do $(sed -i '$d' $NAFN.tex) ; done
cat endi >> $NAFN.tex
emacs --version P head -n1 > innleggemacs
g++ --version P head -n1 > innleggcpp
xelatex --version P head -n1  > innleggxelatex
cweave  --version P head -n1 > innleggcweave
ctangle  --version P head -n1  > innleggctangle
uname  --kernel-release -o  > innleggop
bash --version P head -n1 > innleggbash
sed -i 's/x86/x86\\/g' innleggbash
gsl-config --version > innlegggsl
xelatex $NAFN.tex
```

where `P` is the system pipe operator. Figure 1 records an example of estimates of $\mathbb{E}[R_i(n)]$.

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

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

3 intro

Let $n, r, k_1, \ldots, k_r, \in \mathbb{N}$, $n, k_1, \ldots, k_r \geq 2$, $\sum_i k_i \leq n$, $s = n - \sum_i k_i$. The δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent $\{\xi^n\}$ has transition rate

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

where $0 < \alpha < 1$, $\kappa \geq 2$, $c \geq 0$ all fixed, and

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

and $\kappa + 2 < c_\kappa < \kappa^2$ for $\kappa > 2$. The δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent is an example of a simultaneous multiple-merger coalescent. Simultaneous mergers in up to $\lfloor n/2 \rfloor$ groups for n current number of blocks may occur at such times.

The Ω-δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent is a diploid version of the δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent. In the diploid version, each group of blocks in a 'partition' sampled from the δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent is split into 4 boxes uniformly at random and the blocks in the same box are merged. Since the Ω-δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent can be obtained from a model of a diploid panmictic population evolving absent selfing and according to sweepstakes reproduction the boxes represent the 4 parent chromosomes (2 from each parent) involved in a large offspring number event. With this C++ code we approximate $\mathbb{E}[R_i(n)]$ when the coalescent is the Ω-δ_0-Poisson-Dirichlet$(\alpha, 0)$

We sample the Ω-δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent by *(i)* sampling a partition from the δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent and *(ii)* splitting the partition sizes (blocks) into boxes uniformly at random and with replacement; we repeat steps *(i)* and *(ii)* until we have at least one box with at least 2 blocks. A merger then occurs and we merge blocks and update the tree (the current block sizes).

The algorithm is summarised in § 4, the code follows in § 4.1 – § 4.22; we conclude in § 5. Comments within the code are in **this font and colour**

4 code

we summarise the algorithm;

$$\lambda_m = \sum_{\substack{2\le k_1\le\ldots\le k_r\le m\\ k_1+\cdots+k_r\le m}} \lambda_{m;k_1,\ldots,k_r;s} \tag{3}$$

1. generate all possible (simultaneous ordered) mergers for $m = 2, 3, \ldots, n$ blocks § 4.10
2. for every merger(s) sizes compute and record the transition rate (1)
3. record the total transition rate λ_m (3) for $m = 2, 3, \ldots, m$ for sampling partitions
4. $(r_1, \ldots, r_{n-1}) \leftarrow (0, \ldots, 0)$
5. for each of M experiments : § 4.21
   (a) $(\ell_1, \ldots, \ell_{n-1}) \leftarrow (0, \ldots, 0)$
   (b) $m \leftarrow n$
   (c) $(\xi_1, \ldots, \xi_n) \leftarrow (1, \ldots, 1)$
   (d) **while** $m > 1$: § 4.20
      i. $t \leftarrow \text{Exp}(\lambda_m)$
      ii. $\ell_\xi \leftarrow \ell_\xi + t$ for $\xi = \xi_1, \ldots, \xi_m$
      iii. sample a partition $k_1, \ldots, k_r$ § 4.11 using the transition rates (1) and split partition into boxes § 4.15 until at least one box has at least 2 blocks § 4.19
      iv. given merger sizes merge blocks § 4.18
      v. $m \leftarrow m - \sum_i b_i \mathbb{1}_{\{b_i>1\}} + \sum_i \mathbb{1}_{\{b_i>1\}}$ where b_i is the number of blocks in box i
   (e) $r_i \leftarrow \ell_i / \sum_j \ell_j$ for $i = 1, 2, \ldots, n-1$
6. return r_i/M as an approximation of $\mathbb{E}[R_i(n)]$ for $i = 1, 2, \ldots, n-1$

4.1 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 <boost/math/special_functions/factorials.hpp>
#include "diploid_delta_poisson_dirichlet.hpp"
```

This code is used in chunk 26.

4.2 random number generators

define the STL and GSL random number generators

the STL rngs

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

the GSL random number generator

```
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) ;
}
```

6 ⟨ rngs 6 ⟩ ≡

```
/*
    obtain a seed out of thin air for the random number engine */
std::random_device randomseed;    /*
    Standard mersenne twister random number engine */
std::mt19937_64 rng(randomseed());
gsl_rng * rngtype;
static void setup_rng(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 26.

4.3 descending factorial

compute the descending factorial $(x)_m \equiv x(x-1)\cdots(x-m+1)$ and $(x)_0 \equiv 1$.

using the boost library:

```
static double descending_factorial(const double x, const double m)
{
return static_cast<double>( boost::math::falling_factorial( x,
static_cast<unsigned int>(m)) ) ;
}
```

A direct implementation is

```
double p = 1;
for( double i = 0; i < m; ++i){
p *= (x - i); }
return p ;
```

7 ⟨ compute descending factorial 7 ⟩ ≡

```
static double descending_factorial(const double x, const double m)
{    /*
        using the boost library function :  */
  return static_cast⟨double⟩(boost::math::falling_factorial(x, static_cast⟨unsigned
        int⟩(m)));    /*
        direct implementation :  double p = 1; for (double i = 0; i < m; ++i) {
        p *= (x − i); } return p; */
}
```

This code is used in chunk 26.

4.4 a power function

compute x^y checking for under 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 return( fetestexcept(FE_UNDERFLOW) ? 0. : (fetestexcept(FE_OVERFLOW) ? FLT_MAX
6 : d) ) ;
7 }
```

8 ⟨ checked power function 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 26.

4.5 $\lambda_{n;k_1,\ldots,k_r;s}\ (1)$

compute the rate $\lambda_{n;k_1,\ldots,k_r;s}\ (1)$ of the (ordered) partition $(k_1,\ldots,k_r)$

9 ⟨ compute $\lambda_{n;k_1,\ldots,k_r;s}\ (1)$ 9 ⟩ ≡

```
static double lambdanks (const double n, const std::vector < unsigned int > &v_k ) {
  /*
    v_k is the partition to be split into boxes */
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( ));      /*
    the counts ∑_i 𝟙_{k_i=j} for j = 2,...,n  */
std::unordered_map < unsigned int , unsigned int > counts
{ }
;
for (std::size_t i = 0; i < v_k.size( ); ++i) {      /*
      f = ∏_{i=1}^{r} (k_i − α − 1)_{k_i−1}  */
  f *= descending_factorial(static_cast⟨double⟩(v_k[i]) − 1. − ALPHA,
      static_cast⟨double⟩(v_k[i]) − 1);      /*
      count number of occurrences c_j = ∑_i 𝟙_{k_i=j} of each block in the partition (k_1,...,k_r)
       */
  ++counts[v_k[i]];      /*
      k = k_1 + ··· + k_r  */
  k += static_cast⟨double⟩(v_k[i]);      /*
      d = ∑_i log Γ(k_i + 1)  */
  d += lgamma(static_cast⟨double⟩(v_k[i] + 1));
}
assert(k < n + 1);
const double s = n − k;      /*
    p = ∑_j log Γ(c_j + 1)  */
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);
}
) ) ;
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));
return (veldi(exp(1),
    (lgamma(n + 1.) − d) − lgamma(n − k + 1) − p) * l / (CKAPPA + (CEPS * (1 − ALPHA)))); }
```

This code is used in chunk 26.

4.6 generate all fixed-size partitions summing to a given number

generate all partitions of fixed size *PartitionSize* summing to $myInt \leq m$ and compute the rate $\lambda_{m;k_1,\ldots,k_r;s}$ for it when m is the number of blocks

10 ⟨ partition 10 ⟩ ≡

```
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 ) {
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;
  char a = (idx_Spill) ? ∼((−3 ≫ (LeftRemain − MinVal)) ≪ 2) : 11;
  char b = (−3 ≫ (val_Dec − LeftRemain));
  switch (a & b) {
  case 1: case 2: case 3: idx_Dec = idx_Spill;
    LeftRemain = 1 + (idx_Spill − idx_Dec + 1) ∗ MinVal;
    break;
  case 5:
    for (++idx_Dec, LeftRemain = (idx_Dec − idx_Spill) ∗ val_Dec;
         (idx_Dec ≤ idx_Last) ∧ (partition[idx_Dec] ≤ MinVal); idx_Dec++)
      LeftRemain += partition[idx_Dec];
    LeftRemain += 1 + (idx_Spill − idx_Dec + 1) ∗ MinVal;
    break;
  case 6: case 7: case 11: idx_Dec = idx_Spill + 1;
    LeftRemain += 1 + (idx_Spill − idx_Dec + 1) ∗ MinVal;
    break;
  case 9:
    for (++idx_Dec, LeftRemain = idx_Dec ∗ val_Dec;
         (idx_Dec ≤ idx_Last) ∧ (partition[idx_Dec] ≤ (val_Dec + 1));
         idx_Dec++) LeftRemain += partition[idx_Dec];
    LeftRemain += 1 − (idx_Dec − 1) ∗ MinVal;
    break;
```

```
    case 10:
      for (LeftRemain += idx_Spill * MinVal + (idx_Dec - idx_Spill) * val_Dec + 1, ++idx_Dec;
           (idx_Dec ≤ idx_Last) ∧ (partition[idx_Dec] ≤ (val_Dec - 1)); idx_Dec++)
        LeftRemain += partition[idx_Dec];
      LeftRemain -= (idx_Dec - 1) * MinVal;
      break;
    }
    while (idx_Spill > idx_SpillPrev) partition[--idx_Spill] = MinVal;
    assert(static_cast⟨unsigned int⟩(std::accumulate(partition.begin(), partition.end(),
        0)) ≡ myInt);      /*
        § 4.5 */
    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 the sum of the rates (1) for the generated partitions */
  return sumrates; }
```

This code is used in chunk 26.

4.7 generate all partitions summing to a given number

generate all partitions summing to a given number m when n is the number of blocks

11 ⟨ fixed-sum partitions 11 ⟩ ≡

```
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};      /*
    § 4.5 */
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) {      /*
       s is the size of the partitions summing to m */
  for (unsigned int s = 2; s ≤ m/2; ++s) {
    assert(m > 2 ∗ (s − 1));      /*
        § 4.6 */
    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 26.

4.8 write partitions to file

write partitions for given number of blocks n to file

12 ⟨ file partitions 12 ⟩ ≡

```
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 26.

4.9 generate all partitions when n blocks

generate all partitions when given n blocks

13 ⟨ generate all partitions when n blocks 13 ⟩ ≡

```
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) {      /*
        § 4.7; the partition sums to k; the number of blocks is n */
  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];
}
) ;     /*
    § 4.8 */
ratesmergersfile(n, indx, vlk, v___lambdan[n], a___cmf); }
```

This code is used in chunk 26.

4.10 all partitions

generate all partitions and rates (1) up to sample size

14 ⟨ generate all partitions and rates 14 ⟩ ≡

```
static void all_partitions_rates ( std::vector < double > &vlmn, std::vector < std::vector
    < double >> &acmf )
{
  for (unsigned int tmpn = 2; tmpn ≤ SAMPLE_SIZE; ++tmpn) {      /*
      § 4.9 */
    allmergers_when_n_blocks(tmpn, vlmn, acmf);
  }
}
```

This code is used in chunk 26.

4.11 sample a partition

sample a partition given number of blocks

15 ⟨ sample partition 15 ⟩ ≡

```
static unsigned int samplemerger (const unsigned int n, const std::vector <
    double > &v___cmf )
{    /*
      n the number of blocks; v___cmf the cumulative mass function based on (1)  */
  unsigned int j
  {}
  ;
  const double u = gsl_rng_uniform(rngtype);
  while (u > v___cmf[j]) {
    ++j;
  }
  return j;
}
```

This code is used in chunk 26.

4.12 read partition from file

read a partition from file given the line indicator sampled using § 4.11

16 ⟨ fetch partition from file 16 ⟩ ≡

```
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 26.

4.13 sample a box

sample one of four boxes uniformly; blocks in the same box will be merged

17 ⟨ pick a box 17 ⟩ ≡

```
unsigned int sample_box( )
{
  const double u = gsl_rng_uniform(rngtype);
  return (u < 0.25 ? 0 : (u < .5 ? 1 : (u < .75 ? 2 : 3)));
}
```

This code is used in chunk 26.

4.14 split one partition element

split the blocks in one element of a partition using § 4.13; we sample k iid random variables $I_1, \ldots, I_k$ where $\mathbb{P}(I_i = j) = 1/4$ for $j = 0, 1, 2, 3$

18 ⟨ split partition element into boxes 18 ⟩ ≡

```
void split_blocks (const unsigned int k, std::vector < unsigned int > &split_partition ) {
  /*
    k is the given size of the given partition element */
  std::vector < unsigned int > boxes(4);      /*
    split the k blocks into boxes; boxes[j] = ∑_{i=1}^{k} 𝟙_{I_i=j}  */
  for (unsigned int j = 0; j < k; ++j) {      /*
      § 4.13 */
    ++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) {      /*
      append box sizes with at least 2 blocks to split_partition */
  std::copy_if (boxes.cbegin( ), boxes.cend( ), std::back_inserter(split_partition), [](const auto
        b)
  {
    return b > 1;
  }
  ) ; } }
```

This code is used in chunk 26.

4.15 split a given partition into boxes

split a partition into boxes; given a partition $(k_1, \ldots, k_r)$ split each of k_i blocks into boxes and record the box sizes with at least 2 blocks

19 ⟨ box a given partition 19 ⟩ ≡

```
void split_groups ( const std::vector < unsigned int > &partition, std::vector < unsigned
    int > &split_partition ) {
split_partition.clear( );
for (const auto &k:partition)
{    /*
       § 4.14 */
  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 26.

4.16 update lengths ℓ_i

update realisation ℓ_i of $L_i(n)$; t the interval time until next merger during which have tree configuration (block sizes) *tree*

20 ⟨ given interval time update lengths 20 ⟩ ≡

```
static void update_lengths ( const std::vector < unsigned int > &tree, std::vector <
        double > &v_l, const double t ) {
    for (const auto &b:tree)
    {
      v_l[0] += t;
      v_l[b] += t;
    }
    }
```

This code is used in chunk 26.

4.17 update r_i

update approximations r_i of $\mathbb{E}\left[R_i(n)\right]$ given realised lengths $v\_l\ \ell_1, \ldots, \ell_{n-1}$

21 ⟨ add to r_i 21 ⟩ ≡

```
static void update_ri ( std::vector < double > &v_ri, const std::vector < double > &v_l
    ) {
  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 26.

4.18 update tree

given merger size(s) update tree by merging blocks

22 ⟨ merge blocks and update tree 22 ⟩ ≡

```
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);      /*
        newblocks record the size of the new blocks (obtained by merging blocks
        according to mergers in the current tree) */
    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 26.

4.19 until a merger occurs

the goal here is to

1. sample time and partitions and box sizes until have at least one box with at least 2 blocks;
2. titions and box sizes until have at least one box with at least 2 blocks; then shuffle the tree (the current block sizes)
3. merge blocks by summing block sizes of the rightmost blocks of the shuffled tree given merger sizes
4. remove blocks that have merged from the tree
5. append new blocks from the merged blocks to the tree

23 ⟨ sample time and boxes until merger 23 ⟩ ≡

```
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 ) {
  std::vector < unsigned int > v_partition(SAMPLE_SIZE/2);
  v_partition.reserve(SAMPLE_SIZE/2);
  unsigned int lina
  { }
  ;
  double t
  { }
  ;
  v_merger_sizes.clear( );
  while ( std::all_of (v_merger_sizes.cbegin( ), v_merger_sizes.cend( ), [](const
        auto m)
  {
    return m < 2;
  }
  ) )
  {
    t += gsl_ran_exponential(rngtype, 1./v_lambdan[current_number_blocks]);
      /*
        § 4.11 */
    lina = samplemerger(current_number_blocks, v___cmf);      /*
        § 4.12 */
    readmergersizes(current_number_blocks, 1 + lina, v_partition);      /*
        § 4.15 */
    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 26.

4.20 one experiment

generate one realisation of $\ell_1, \ldots, \ell_{n-1}$ starting with tree configuration $(1, \ldots, 1)$ (all blocks of size 1) and ending with one block of size n

24 ⟨ generate one realisation of $\ell_1, \ldots, \ell_{n-1}$ 24 ⟩ ≡

```
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
  { }
  ;
  std::vector < unsigned int > v___merger_sizes
  { }
  ;
  v___merger_sizes.clear( );
  v___merger_sizes.reserve(2 * SAMPLE_SIZE);
  while (tree.size( ) > 1) {      /*
      § 4.19 */
    t = until_merger(tree.size( ), v___lambdan, a___cmf[tree.size( )], v___merger_sizes);
    /*
      § 4.16 */
    update_lengths(tree, v_l, t);      /*
      § 4.18 */
    update_tree(tree, v___merger_sizes);
  }
  assert(tree.back( ) ≡ SAMPLE_SIZE);      /*
    § 4.17 */
  update_ri(v_r, v_l); }
```

This code is used in chunk 26.

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

approximate $\mathbb{E}[R_i(n)]$ for a given number of EXPERIMENTS

25 ⟨ go ahead – approximate $\mathbb{E}[R_i(n)]$ 25 ⟩ ≡

```
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 > { } ) ;      /*
        § 4.10 */
    all_partitions_rates(v___lambdan, a___cmfs);
    int r = EXPERIMENTS + 1;
    while (--r > 0) {      /*
          § 4.20 */
      one_experiment(v___l, vri, v___lambdan, a___cmfs);
    }
    for (const auto &z:vri)
    {
      std::cout ≪ z ≪ '\n';
    }
    }
```

This code is used in chunk 26.

4.22 main

the *main* module

```
26      /*
        § 4.1 */
    ⟨ includes 5 ⟩    /*
        § 4.2 */
    ⟨ rngs 6 ⟩    /*
        § 4.3 */
    ⟨ compute descending factorial 7 ⟩    /*
        § 4.4 */
    ⟨ checked power function 8 ⟩    /*
        § 4.5 */
    ⟨ compute λ_{n;k_1,...,k_r;s} (1) 9 ⟩    /*
        § 4.6 */
    ⟨ partition 10 ⟩    /*
        § 4.7 */
    ⟨ fixed-sum partitions 11 ⟩    /*
        § 4.8 */
    ⟨ file partitions 12 ⟩    /*
        § 4.9 */
    ⟨ generate all partitions when n blocks 13 ⟩    /*
        § 4.10 */
    ⟨ generate all partitions and rates 14 ⟩    /*
        § 4.11 */
    ⟨ sample partition 15 ⟩    /*
        § 4.12 */
    ⟨ fetch partition from file 16 ⟩    /*
        § 4.13 */
    ⟨ pick a box 17 ⟩    /*
        § 4.14 */
    ⟨ split partition element into boxes 18 ⟩    /*
        § 4.15 */
    ⟨ box a given partition 19 ⟩    /*
        § 4.16 */
    ⟨ given interval time update lengths 20 ⟩    /*
        § 4.17 */
    ⟨ add to r_i 21 ⟩    /*
        § 4.18 */
    ⟨ merge blocks and update tree 22 ⟩    /*
        § 4.19 */
    ⟨ sample time and boxes until merger 23 ⟩    /*
        § 4.20 */
    ⟨ generate one realisation of ℓ_1, ..., ℓ_{n-1} 24 ⟩    /*
        § 4.21 */
    ⟨ go ahead – approximate 𝔼[R_i(n)] 25 ⟩
    int main(int argc, const char *argv[])
    {    /*
            § 4.2 */
      setup_rng(static_cast⟨std::size_t⟩(atoi(argv[1])));    /*
            § 4.21 */
      approximate_eri( );
      gsl_rng_free(rngtype);
      return 0;
```

```
}
```

5 conclusions

We approximate $\mathbb{E}[R_i(n)]$ for the Ω-δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent with $0 < \alpha < 1$. Figure 1 records an example.

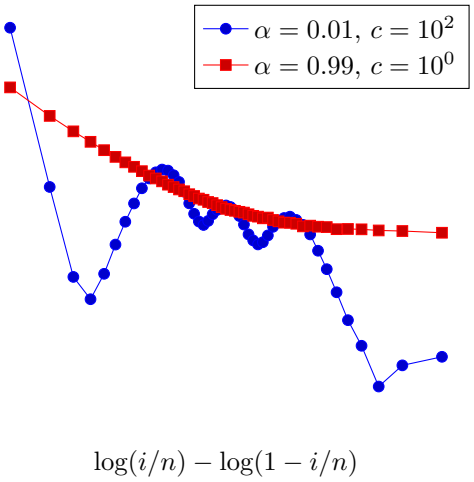

Figure 1: *An example approximation of* $\mathbb{E}[R_i(n)]$ *graphed as logits as a function of* $\log(i/n) - \log(1 - i/n)$ *for* $i = 1, 2, \ldots, n-1$ *where* n *is sample size*

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

[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

⟨ add to r_i 21 ⟩ Used in chunk 26.
⟨ box a given partition 19 ⟩ Used in chunk 26.
⟨ checked power function 8 ⟩ Used in chunk 26.
⟨ compute $\lambda_{n;k_1,\ldots,k_r;s}\ (1)$ 9 ⟩ Used in chunk 26.
⟨ compute descending factorial 7 ⟩ Used in chunk 26.
⟨ fetch partition from file 16 ⟩ Used in chunk 26.
⟨ file partitions 12 ⟩ Used in chunk 26.
⟨ fixed-sum partitions 11 ⟩ Used in chunk 26.
⟨ generate all partitions and rates 14 ⟩ Used in chunk 26.
⟨ generate all partitions when n blocks 13 ⟩ Used in chunk 26.
⟨ generate one realisation of $\ell_1, \ldots, \ell_{n-1}$ 24 ⟩ Used in chunk 26.
⟨ given interval time update lengths 20 ⟩ Used in chunk 26.
⟨ go ahead – approximate $\mathbb{E}\left[R_i(n)\right]$ 25 ⟩ Used in chunk 26.
⟨ includes 5 ⟩ Used in chunk 26.
⟨ merge blocks and update tree 22 ⟩ Used in chunk 26.
⟨ partition 10 ⟩ Used in chunk 26.
⟨ pick a box 17 ⟩ Used in chunk 26.
⟨ rngs 6 ⟩ Used in chunk 26.
⟨ sample partition 15 ⟩ Used in chunk 26.
⟨ sample time and boxes until merger 23 ⟩ Used in chunk 26.
⟨ split partition element into boxes 18 ⟩ Used in chunk 26.