Gene genealogies in haploid populations evolving according to sweepstakes reproduction – approximating $\mathbb{E}[R_i(n)]$ for the δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent

Bjarki Eldon[1]

Let $[n] \equiv \{1, 2, \ldots, n\}$ for any $n \in \mathbb{N} \equiv \{1, 2, \ldots\}$. Write $\{\xi^n\} \equiv \{\xi^n(t); t \geq 0\}$ for the δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent. Write $\#A$ for the number of elements in a given (finite) set A, $\tau(n) \equiv \inf\{t \geq 0 : \#\xi^n(t) = 1\}$, $L_i(n) \equiv \int_0^{\tau(n)} \#\{\xi \in \xi^n(t) : \#\xi = i\}\, dt$ for all $i \in [n-1]$, $L(n) \equiv \int_0^{\tau(n)} \#\xi^n(t) dt$. Then $L(n) = L_1(n) + \cdots + L_{n-1}(n)$; write $R_i(n) \equiv L_i(n)/L(n)$. With this C++ simulation code we approximate $\mathbb{E}[R_i(n)]$.

Contents

[1] beldon11@gmail.com
compiled @ 11:18am on Tuesday 6th January, 2026; using
6.17.13+deb14-amd64 GNU/Linux
GNU bash, version 5.3.3(1)-release (x86_64-pc-linux-gnu)
g++ (Debian 15.2.0-12) 15.2.0
CTANGLE 4.12.1 (TeX Live 2025/Debian)
CWEAVE 4.12.1 (TeX Live 2025/Debian)
XƎLaTeX XeTeX 3.141592653-2.6-0.999997 (TeX Live 2025/Debian)
SpiX 1.3.0
GNU Awk 5.3.2, API 4.0, PMA Avon 8-g1, (GNU MPFR 4.2.2, GNU MP 6.3.0)
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 %$)

```
%$ NAFN=delta_poisson_dirichlet_usingrates
%$ echo 'constexpr unsigned int SAMPLE_SIZE = 20;' > $NAFN.hpp
%$ echo 'constexpr double ALPHA = 0.05;' >> $NAFN.hpp
%$ echo 'constexpr double KAPPA = 2.0 ;' >> $NAFN.hpp
%$ echo 'constexpr double CEPS = 1. ;' >> $NAFN.hpp
%$ echo 'constexpr double MINF = (2. + (1 + pow(2., 1-KAPPA)/(KAPPA - 1)))/2.
%;' >> $NAFN.hpp
%$ echo 'constexpr double CKAPPA = 2/pow(MINF,2.) ;' >> $NAFN.hpp
%$ echo 'constexpr int EXPERIMENTS = 10000 ;' >> $NAFN.hpp
%$ ctangle $NAFN.w
%$ g++ -std=c++26 -m64 -march=native -O3 -x c++ $NAFN.c -lm -lgsl -lgslcblas
%$ rm -f gg_*_.txt
%$ ./a.out $(shuf -i 43484-2392022 -n1) > logitresout
%$ seq 19 P awk '{S=20;print log($1/S) - log(1 - ($1/S))}' > nlogits
%$ paste -d',' nlogits logitresout > forplottingfile1
%$ sed -i 's/ALPHA = 0.05/ALPHA = 0.95/g' $NAFN.hpp
%$ ctangle $NAFN.w
%$ g++ -std=c++26 -m64 -march=native -O3 -DNDEBUG -x c++ $NAFN.c -lm -lgsl
%-lgslcblas
%$ rm -f gg_*_.txt
%$ ./a.out $(shuf -i 43484-2392022 -n1) > logitresout
%$ paste -d',' nlogits logitresout > forplottingfile2
%$ emacs --version P head -n1 > innleggemacs
%$ g++ --version P head -n1 > innleggcpp
%$ xelatex --version P head -n1  > innleggxelatex
%$ cweave  --version P head -n1 > innleggcweave
%$ uname  --kernel-release -o  > innleggop
%$ bash --version P head -n1 > innleggbash
%$ sed -i 's/x86/x86\\/g' innleggbash
%$ NAFN=delta_poisson_dirichlet_usingrates
%$ cweave $NAFN.w
%$ tail -n4 $NAFN.tex > endi
%$ for i in $(seq 5); do $(sed -i '$d' $NAFN.tex) ; done
%$ cat endi >> $NAFN.tex
%$ 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

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

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. It allows simultaneous mergers in up to $\lfloor n/2 \rfloor$ groups for any given n.

In § 4 we summarise the algorithm, the code follows in § 4.1 – § 4.18, we conclude in § 5. Comments within the code are in **this font and colour**

4 code

we summarise the algorithm; λ_m denotes the total transition rate

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

```
1. generate all possible (simultaneous) mergers for $m = 2, 3, \ldots, n$ blocks
2. for every merger(s) sizes compute and record the transition rate (1)
3. record the total transition rate $\lambda_m$ (3) for $m = 2, 3, \ldots, m$
4. $(r_1, \ldots, r_{n-1}) \leftarrow (0, \ldots, 0)$
5. for each of $M$ experiments :
   (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** $m > 1$ :
       i. $t \leftarrow \mathrm{Exp}(\lambda_m)$
       ii. $\ell_\xi \leftarrow \ell_\xi + t$ for $\xi = \xi_1, \ldots, \xi_m$
       iii. sample merger sizes $k_1, \ldots, k_r$ using the transition rates
       iv. given merger sizes merge blocks
       v. $m \leftarrow m - \sum_i k_i + r$
   (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

5 ⟨includes 5⟩ ≡

```
#include <iostream>
#include <cstdlib>
#include <iterator>
#include <random>
#include <fstream>
#include <iomanip>
#include <vector>
#include <numeric>
#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 "./delta_poisson_dirichlet_usingrates_wplotting.hpp"
```

This code is used in chunk 22.

4.2 the random number generators

the STL and GSL random number generators

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 seeded with rng() */
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 22.

4.3 descending factorial

the descending factorial $(x)_m \equiv x(x-1)\cdots(x-m+1)$ with $(x)_0 \equiv 1$; recall (2)

7 $\langle$ descending factorial 7 $\rangle \equiv$

```
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 22.

4.4 x^y

compute x^y guarding against over- and underflow

8 ⟨ guarded x^y module 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 22.

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

compute $\lambda_{n;k_1,\ldots,k_r;s}$ (1); see the header file for value of `ALPHA`, `KAPPA`, `CEPS`, and `CKAPPA` as in (2)

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

```
static double lambdanks (const double n, const std::vector < unsigned int > &v_k ) {
    /*
     n is the given number of blocks; v_k records the given merger sizes k_1, ..., k_r   */
  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) {
    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;       /*
       p = ∑_j log Γ(1 + ∑_i 𝟙_{k_i=j})   */
  const double p = static_cast⟨double⟩ ( std::accumulate (counts.begin( ), counts.end( ), 0,
            [](double a, const auto &x)
  {
    return a + lgamma(x.second + 1);
  }
  ) ) ;      /*
       veldi § 4.4 */
  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));       /*
       ℓ = 𝟙_{r=1,k_1=2} C_κ + cα^{r+s−1}(Γ(r + s)/Γ(n)) * ∏_{i=1}^r (k_i − 1 − α)_{k_i−1}  */
  return (veldi(exp(1),
       (lgamma(n + 1.) − d) − lgamma(n − k + 1) − p) * l/(CKAPPA + (CEPS * (1 − ALPHA)))); }
```

This code is used in chunk 22.

4.6 r mergers summing to m

generate all ordered r merger sizes $2 \le k_1 \le \ldots \le k_r \le n$ where $\sum_i k_i = myInt$ and r is fixed; for example for $myInt = 6$ and $r = 2$ we should get $(2, 4)$ and $(3, 3)$.

10 $\langle$ r-merger summing to m 10 $\rangle \equiv$

```
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 ) {      /*
      compute all PartitionSize ordered merger sizes summing 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 {      /∗ Value AFTER decrementing ∗/
  unsigned int val_Dec = partition[idx_Dec] − 1;
     /∗ Decrement at the Decrement Point ∗/
  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];
```

```
        $LeftRemain \mathrel{+}= 1 - (idx\_Dec - 1) * MinVal$;
        **break**;
      **case** 10:
        **for** ($LeftRemain \mathrel{+}= idx\_Spill * MinVal + (idx\_Dec - idx\_Spill) * val\_Dec + 1, {+}{+}idx\_Dec$;
              $(idx\_Dec \leq idx\_Last) \wedge (partition[idx\_Dec] \leq (val\_Dec - 1))$; $idx\_Dec{+}{+}$)
          $LeftRemain \mathrel{+}= partition[idx\_Dec]$;
        $LeftRemain \mathrel{-}= (idx\_Dec - 1) * MinVal$;
        **break**;
      }
      **while** $(idx\_Spill > idx\_SpillPrev)$ $partition[{-}{-}idx\_Spill] = MinVal$;
      $assert($**static_cast**$\langle$**unsigned int**$\rangle(std::accumulate(partition.begin(\,), partition.end(\,),$
          $0)) \equiv myInt)$;      /*
          **compute** $\lambda_{m;k_1,\ldots,k_r;s}$ (1) **for the given merger size(s)** *lambdanks* **§ 4.5**  */
      $lrate = lambdanks($**static_cast**$\langle$**double**$\rangle(m), partition)$;
      $assert(lrate \geq 0)$;
      $v\_l\_k.push\_back(std::make\_pair(lrate, partition))$;
      $lrates\_sorting.push\_back(lrate)$;
      $sumrates \mathrel{+}= lrate$;
    } **while** $(idx\_Dec \leq idx\_Last)$;
    $assert(sumrates \geq 0)$;      /*
        **return the sum of the** *PartitionSize* **merger sizes summing to** *myInt*  */
    **return** $sumrates$; }
```
This code is used in chunk 22.

4.7 merger sizes summing to m

generate all ordered merger sizes summing to a given $m \leq n$ when n is the given number of blocks

11 ⟨merger sizes suming to m 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 ) {     /*
      generate all ordered merger sizes summing to m   */
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) {
   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 22.

4.8 merger sizes to file

print merger sizes to file

12 ⟨print merger sizes to file 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) {      /*
                 record the corresponding cmf  */
            cmf += (vlk[i].first)/s;
            assert(cmf ≥ 0);      /*
                  possibly write also the corresponding cmf to file; f « cmf « ' ' ;  */
            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 22.

4.9 all merger sizes when m blocks

generate all possible ordered merger sizes when m blocks

13 ⟨possible merger sizes when m 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 */
  lambdan += allmergers_sum_m(n, k, vlk, ratetosort);
}      /*
        record the total rate for each given number of blocks; use for sampling time
        between mergers  */
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 22.

4.10 all possible merger sizes given sample size

generate all possible merger sizes given sample size `SAMPLE_SIZE`

14 ⟨allmergers 14⟩ ≡

```
static void allmergers ( 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 22.

4.11 update tree

merge blocks and update the current tree configuration, the current block sizes. given merger sizes $k_1, \ldots, k_r$, $k = \sum_i k_i$, and the current tree with block sizes $(\xi_1, \ldots, \xi_m)$ where $\xi_j \in [n]$ and $\sum_j \xi_j = n$ with m blocks the goal here is to

$$\begin{aligned}
\text{randomize the blocks: } & \big(\xi_{\sigma(1)}, \ldots, \xi_{\sigma(m)}\big) \\
\text{merge blocks: } & \left(\xi_{\sigma(1)}, \ldots, \xi_{\sigma(m-k)}, \quad \underbrace{\xi, \ldots, \xi}_{\xi'_r \text{ is sum of } k_r \text{ blocks}} \quad , \ldots , \quad \underbrace{\xi, \ldots, \xi_{\sigma(m)}}_{\xi'_1 \text{ is sum of } k_1 \text{ blocks}} \right) \\
\text{return the new tree: } & \big(\xi_{\sigma(1)}, \ldots, \xi_{\sigma(m-k)}, \xi'_r, \ldots, \xi'_1\big)
\end{aligned}$$

15 ⟨update the tree configuration 15⟩ ≡

```
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);
    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 22.

4.12 update lengths

given current block sizes $\xi_1, \ldots, \xi_m$ of the current m blocks the goal here is to

1. sample a random exponential time with rate λ_m
2. $\ell_\xi \leftarrow \ell_\xi + t$ for $\xi = \xi_1, \ldots, \xi_m$

where ℓ_ξ a realisation of $L_\xi(n)$

16 ⟨update $\ell_i(n)$ 16⟩ ≡

```
static void updatelengths ( const std::vector < unsigned int > &tre, std::vector <
        double > &v___lengths, const std::vector < double > &v_lambdan ) {
    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 22.

4.13 update r_i

update the approximation of $\mathbb{E}\left[R_i(n)\right]$; given lengths $\ell_1, \ldots, \ell_{n-1}$ the goal here is to

$$r_i \leftarrow r_i + \frac{\ell_i}{\ell_1 + \cdots + \ell_{n-1}}$$

for $i = 1, 2, \ldots, n-1$

17 $\langle$ update r_i 17 $\rangle \equiv$

```
static void updateri ( const std::vector < double > &v___l, std::vector < double > &v_ri
      ) {      /*
     v___l[0] is the sum $\sum_i \ell_i$   */
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 22.

4.14 sample merger sizes

sample merger size(s) $(k_1, \ldots, k_r)$ by returning $\inf\{j \in \{0,1,2\} : F_j > u\}$ where u is a random uniform and F_j a cumulative mass function generated by ordering the merger size(s) in descending order by the rate $\lambda_{n;k_1,\ldots,k_r;s}$ (1)

18 ⟨ get merger size(s) 18 ⟩ ≡

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

This code is used in chunk 22.

4.15 read merger size(s) from file

read the merger sizes(s) corresponding to the index sampled in § 4.14

19 ⟨read in merger size(s) from file 19⟩ ≡

```
  static void readmergersizes (const unsigned int n, const unsigned int j, std::vector <
      unsigned int > &v___mergers ) {      /*
      n the current number of blocks; j the index sampled using § 4.14; v___mergers will
      record the merger size(s)  */
  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 22.

4.16 one realisation of $(L_1(n), \ldots, L_{n-1}(n))$

get one realisation of $(L_1(n), \ldots, L_{n-1}(n))$; given $(\ell_1, \ldots, \ell_{n-1})$ update $(r_1, \ldots, r_{n-1})$
the goal here is

1. **while** at least two blocks :
   (a) sample time until merger and update $\ell_i \leftarrow \ell_i + t$ § 4.12
   (b) sample merger size(s) $k_1, \ldots, k_r$ § 4.14
   (c) merge blocks and update tree § 4.11
2. update $r_i \leftarrow r_i + \ell_i / \sum_j \ell_j$ § 4.13

20 ⟨get $(\ell_1, \ldots, \ell_{n-1})$ 20⟩ ≡

```
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 ) {      /*
             v___ri holds the approximations r_1,...,r_{n-1};  vl holds ℓ_1,...,ℓ_{n-1};
            v___lambda___n the total rates λ_n;  a___cmf the cumulative mass functions for
            sampling merger size(s) */      /*
            initialise the block sizes */
        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) {      /*
              § 4.12  */
          updatelengths(v___tre, vl, v___lambda___n);      /*
              § 4.14  */
          lina = samplemerger(v___tre.size(), a___cmf[v___tre.size()]);      /*
              § 4.15 */
          readmergersizes(v___tre.size(), 1 + lina, v_merger_sizes);      /*
              § 4.11 */
          updatetree(v___tre, v_merger_sizes);
        }
        assert(v___tre.size() < 2);
        assert(v___tre.back() ≡ SAMPLE_SIZE);      /*
            § 4.13 */
        updateri(vl, v___ri); }
```

This code is used in chunk 22.

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

approximate $\mathbb{E}[R_i(n)]$ as predicted by the δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent with transition rates (1) given sample size n; the goal here is

1. given sample size generate all possible merger sizes and corresponding transition rates § 4.10
2. for each of $M =$ `EXPERIMENTS` experiments :
   (a) sample a realisation $\ell_1, \ldots, \ell_{n-1}$ of $L_1(n), \ldots, L_{n-1}(n)$ § 4.16
   (b) update $r_i \leftarrow r_i + \ell_i / \sum_j \ell_j$ § 4.13
3. return r_i/M as approximation of $\mathbb{E}[R_i(n)]$ for $i = 1, 2, \ldots, n-1$

21 ⟨ go ahead – approximate $\mathbb{E}[R_i(n)]$ 21 ⟩ ≡

```
static void approximate( ) {      /*
        vri holds r_1,...,r_{n-1};  v___l holds ℓ_1,...,ℓ_{n-1};  v_l_n holds λ_n for
        n = 2,...,SAMPLE_SIZE;  a___cmfs holds the cumulative mass functions for sampling
        merger size(s)  */
    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 > { } ) ;      /*
        § 4.10  */
    allmergers(v_l_n, a___cmfs);
    int r = EXPERIMENTS + 1;
    while (−−r > 0)  {      /*
          § 4.16  */
      onexperiment(vri, v___l, v_l_n, a___cmfs);
    }
    for (unsigned int i = 1;  i < SAMPLE_SIZE;  ++i)  {
      std::cout ≪ log(vri[i]/static_cast⟨double⟩(EXPERIMENTS)) − log(1. −
          (vri[i]/static_cast⟨double⟩(EXPERIMENTS))) ≪ '\n';
    }
    }
```

This code is used in chunk 22.

4.18 main

the *main* module; here we

1. initialise the GSL random number generator § 4.2
2. approximate $\mathbb{E}[R_i(n)]$ § 4.17

22
```
    /*
    § 4.1 */
⟨ includes 5 ⟩    /*
        § 4.2 */
    ⟨ rngs 6 ⟩    /*
        § 4.3 */
    ⟨ descending factorial 7 ⟩    /*
        § 4.4 */
    ⟨ guarded $x^y$ module 8 ⟩    /*
        § 4.5 */
    ⟨ compute $\lambda_{n;k_1,\ldots,k_r;s}$ 9 ⟩    /*
        § 4.6 */
    ⟨ $r$-merger summing to $m$ 10 ⟩    /*
        § 4.7 */
    ⟨ merger sizes suming to $m$ 11 ⟩    /*
        § 4.8 */
    ⟨ print merger sizes to file 12 ⟩    /*
        § 4.9 */
    ⟨ possible merger sizes when $m$ blocks 13 ⟩    /*
        § 4.10 */
    ⟨ allmergers 14 ⟩    /*
        § 4.11 */
    ⟨ update the tree configuration 15 ⟩    /*
        § 4.12 */
    ⟨ update $\ell_i(n)$ 16 ⟩    /*
        § 4.13 */
    ⟨ update $r_i$ 17 ⟩    /*
        § 4.14 */
    ⟨ get merger size(s) 18 ⟩    /*
        § 4.15 */
    ⟨ read in merger size(s) from file 19 ⟩    /*
        § 4.16 */
    ⟨ get $(\ell_1, \ldots, \ell_{n-1})$ 20 ⟩    /*
        § 4.17 */
    ⟨ go ahead – approximate $\mathbb{E}[R_i(n)]$ 21 ⟩
    int main(int argc, char *argv[])
    {    /*
          §4.2 */
      setup_rng(static_cast⟨unsigned long int⟩(atoi(argv[1])));    /*
          § 4.17 */
      approximate( );
      gsl_rng_free(rngtype);
      return 0;
    }
```

5 conclusions and bibliography

We approximate $\mathbb{E}[R_i(n)]$ when associated with the δ_0-Poisson-Dirichlet$(\alpha, 0)$ coalescent with transition rates (1); see § 4 for a summary of the algorithm. Figure 1 records an example.

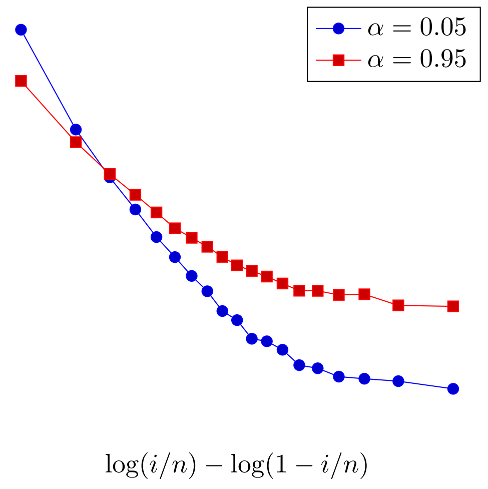

Figure 1: *An example approximation of* $\mathbb{E}[R_i(n)]$ *graphed as logits as a function of* $\log(i/n) - \log(1 - i/n)$ *for* $i = 1, 2, \ldots, n-1$ *where* n *is sample size*

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. Beta-coalescents when sample size is large. https://doi.org/10.64898/2025.12.30.697022

[DK99] P Donnelly and T G Kurtz. Particle representations for measure-valued population models. *Ann Probab*, 27:166–205, 1999.
https://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

$\langle$ r-merger summing to m 10 $\rangle$ Used in chunk 22.
$\langle$ allmergers 14 $\rangle$ Used in chunk 22.
$\langle$ compute $\lambda_{n;k_1,\ldots,k_r;s}$ 9 $\rangle$ Used in chunk 22.
$\langle$ descending factorial 7 $\rangle$ Used in chunk 22.
$\langle$ get $(\ell_1, \ldots, \ell_{n-1})$ 20 $\rangle$ Used in chunk 22.
$\langle$ get merger size(s) 18 $\rangle$ Used in chunk 22.
$\langle$ go ahead – approximate $\mathbb{E}[R_i(n)]$ 21 $\rangle$ Used in chunk 22.
$\langle$ guarded x^y module 8 $\rangle$ Used in chunk 22.
$\langle$ includes 5 $\rangle$ Used in chunk 22.
$\langle$ merger sizes suming to m 11 $\rangle$ Used in chunk 22.
$\langle$ possible merger sizes when m blocks 13 $\rangle$ Used in chunk 22.
$\langle$ print merger sizes to file 12 $\rangle$ Used in chunk 22.
$\langle$ read in merger size(s) from file 19 $\rangle$ Used in chunk 22.
$\langle$ rngs 6 $\rangle$ Used in chunk 22.
$\langle$ update $\ell_i(n)$ 16 $\rangle$ Used in chunk 22.
$\langle$ update r_i 17 $\rangle$ Used in chunk 22.
$\langle$ update the tree configuration 15 $\rangle$ Used in chunk 22.