Gene genealogies in haploid populations evolving according to sweepstakes reproduction — exact $\mathbb{E}[L_i(n)]$ for the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent

Bjarki Eldon[1] [2]

Let $\{\xi^n\} \equiv \{\xi^n(t); t \geq 0\}$ denote the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent for $0 < \gamma \leq 1$ and $0 < \alpha < 2$. Write $L_i(n) \equiv \int_0^{\tau(n)} \#\{\xi \in \xi^n(t) : \#\xi = i\}\, dt$ for $i = 1, \ldots, n-1$, $L(n) \equiv L_1(n) + \cdots + L_{n-1}(n)$, and $\tau(n) \equiv \inf\{t \geq 0 : \#\xi^n(t) = 1\}$. With this C code we compute $\mathbb{E}[L_i(n)]$ predicted by the δ_0-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent. This family of multiple-merger coalescents can be obtained from a model of a haploid panmictic population of constant size evolving according to sweepstakes reproduction (heavy right-tailed offspring number distribution).

Contents

1 Copyright

[1] beldon11@gmail.com

[2] compiled @ 10:59am on Tuesday 6th January, 2026
6.17.13+deb14-amd64 GNU/Linux
CTANGLE 4.12.1 (TeX Live 2025/Debian)
gcc (Debian 15.2.0-12) 15.2.0
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
GNU Awk 5.3.2, API 4.0, PMA Avon 8-g1, (GNU MPFR 4.2.2, GNU MP 6.3.0)
SpiX 1.3.0
GNU Emacs 30.2

2 Compilation, output and execution

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] file.

One can use `cweave` to generate a `.tex` file, and `ctangle` to generate a `.c` file. To compile the C++ code (the `.c` file), one needs the GNU Scientific Library. Using a Makefile can be helpful, naming this file `iguana.w`

```
iguana.pdf :  iguana.tex
     cweave iguana.w
     pdflatex iguana
     bibtex iguana
     pdflatex iguana
     pdflatex iguana
     ctangle iguana
     gcc -std=c23 -O3 -march=native -m64 -x c iguana.c -lm -lgsl -lgslcblas

clean :
     rm -vf iguana.c iguana.tex
```

Use `valgrind` to check for memory leaks:

```
valgrind -v --leak-check=full --show-leak-kinds=all -vgdb=full -leak-resolution=high
-num-callers=50 <program call>
```

Use `cppcheck` to check the code:

```
cppcheck --enable=all --language=c iguana.c
```

Use `splint` to check the code:

3 intro

Suppose the population evolves as in Definition 3.1.

Definition 3.1 (Evolution of a haploid panmictic population) *Consider a haploid, i.e. each individual carries exactly one gene copy, panmictic population of constant size N evolving in discrete (non-overlapping) generations. In any given generation each individual independently produces a random number of potential offspring according to some given law. If the total number of potential offspring produced in this way is at least N, then N of them sampled uniformly at random without replacement form a new set of reproducing individuals, and the remaining potential offspring perish. Otherwise we will assume an unchanged population over the generation (all the potential offspring perish).*

Let $\{\pi_{N,n}(t); t \geq 0\}$ denote the Markov sequence describing the random ancestral relations of n gene copies (leaves) sampled from a population of size N evolving according to Definition 3.1. If the random number of potential offspring (X) of an arbitrary individual is distributed according to

$$\lim_{x\to\infty} Cx^{\alpha}\mathbb{P}\left(X \geq x\right) = 1 \tag{1}$$

[Sch03, Eq 11] then $\{\pi_{N,n}(\lfloor t/c_N \rfloor); t \geq 0\}$ converges as $N \to \infty$ in the J_1-Skorohod topology to the Beta$(2-\alpha, \alpha)$-coalescent provided $1 \leq \alpha < 2$ [Sch03, Thm 4c].

Suppose X is distributed according to

$$g(k)\left(\frac{1}{k^{\alpha}} - \frac{1}{(1+k)^{\alpha}}\right) \leq \mathbb{P}\left(X = k\right) \leq f(k)\left(\frac{1}{k^{\alpha}} - \frac{1}{(1+k)^{\alpha}}\right), \quad k \in [\zeta(N)], \tag{2}$$

where the functions g, f are independent of N and so that $\mathbb{E}\left[X\right] > 1$ and $\mathbb{P}\left(X \leq \zeta(N)\right) = 1$. Furthermore, the population evolves according to Definition 3.2 or Definition 3.3.

Definition 3.2 (The random environment of type A) *Suppose a population evolves according to Definition 3.1 with the number of potential offspring produced by each individual distributed according to (1) or (2). Fix $0 < \alpha < 2 \leq \kappa$. Write E for the event $\{X_i \sim L(\alpha, \zeta(N))$ for all $i \in [N]\}$ and E^{c} when κ replaces α in E for some given $\zeta(N)$. Suppose, with $(\varepsilon_N)_N$ a positive sequence with $0 < \varepsilon_N < 1$ and $\varepsilon_N \to 0$,*

$$\mathbb{P}\left(E\right) = \varepsilon_N, \quad \mathbb{P}\left(E^{\mathsf{c}}\right) = 1 - \varepsilon_N$$

Definition 3.3 (The random environment of type B) *Suppose a population evolves according to Definition 3.1 with the law for the number of potential offspring given by Eq (2). Fix $0 < \alpha < 2 \leq \kappa$. Write E_1 for the event $\{$there exists exactly one $i \in [N] : X_i \sim L(\alpha, \zeta(N)), X_j \sim L(\kappa, \zeta(N))$ for all $j \in [N] \setminus \{i\}\}$ with i picked uniformly at random, and E_1^{c} when κ replaces α in E_1 so that $E_1^{\mathsf{c}} = E^{\mathsf{c}}$ from Definition 3.2. Suppose*

$$\mathbb{P}\left(E_1\right) = \overline{\varepsilon}_N, \quad \mathbb{P}\left(E_1^{\mathsf{c}}\right) = 1 - \overline{\varepsilon}_N$$

where $(\overline{\varepsilon}_N)_N$ is a positive sequence with $0 < \overline{\varepsilon}_N < 1$ for all N.

Then $\{\pi_{N,n}(\lfloor t/c_N \rfloor); t \geq 0\}$ converges as $N \to \infty$ in the J_1-Skorohod topology to the Kingman-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent provided $\liminf_{N\to\infty} \zeta(N)/N > 0$.

Let $L_i(n)$ denote the random length of branches supporting $i \in \{1, 2, \ldots, n-1\}$ leaves; write $L(n) = \sum_i L_i(n)$. We compute

$$\varphi_i(n) := \frac{\mathbb{E}\left[L_i(n)\right]}{\mathbb{E}\left[L(n)\right]} \tag{3}$$

by adapting a recursion for computing $\mathbb{E}\left[L_i(n)\right]$ for Λ-coalescents [BBE2013a]. We compute $\varphi_i(n)$

for the Kingman-Beta$(\gamma, 2-\alpha, \alpha)$-coalescent with transition rates

$$\lambda_{n,k} = \mathbb{1}_{\{k=2\}}\binom{n}{k}\frac{C_\kappa}{C} + \binom{n}{k}\frac{\alpha c'_\alpha}{C m_\infty^\alpha} B(\gamma, k-\alpha, n-k+\alpha) \tag{4a}$$

$$\gamma = \mathbb{1}_{\left\{\frac{\zeta(N)}{N}\to K\right\}}\frac{K}{m_\infty + K} + \mathbb{1}_{\left\{\frac{\zeta(N)}{N}\to\infty\right\}} \tag{4b}$$

$$C = C_\kappa + \frac{\alpha c}{m_\infty^\alpha}\int_{\mathbb{1}_{\{\zeta(N)/N\to K\}}\frac{m_\infty}{K+m_\infty}}^1 (1-y)^{1-\alpha} y^{\alpha-1} dy \tag{4c}$$

$$C_\kappa = 2m_\infty^{-2}\mathbb{1}_{\{\kappa=2\}} + 2m_\infty^{-2}\frac{c_\kappa}{2^\kappa(\kappa-2)(\kappa-1)}\mathbb{1}_{\{\kappa>2\}} \tag{4d}$$

The range of α in Definition 3.2 and Definition 3.3 intersects at 1 so one must decide which of the two definitions holds when $\alpha = 1$. Let where $0 < c' < 1$ and $c > 0$. Write $c'_\alpha = c$ if Definition 3.2 holds when $\alpha = 1$, and $c'_\alpha = \mathbb{1}_{\{\alpha=1\}}c' + \mathbb{1}_{\{\alpha\in(0,2)\setminus\{1\}\}}c$ when Definition 3.3 holds when $\alpha = 1$ (and $\kappa > 2$). We approximate m_∞ with $m_\infty \approx (2 + (1 + 2^{1-\kappa})/(\kappa - 1))/2$. When $\kappa > 2$

$$\kappa + 2 < c_\kappa < \kappa^2$$

4 code

The code requires the GSL Library. Comments within the code are in **this font and colour**; the code follows in § 4.1–§ 4.13, we conclude in § 5.

4.1 the included libraries

the included C libraries

5 ⟨ includes 5 ⟩ ≡

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <assert.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_sf.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fit.h>
#include <gsl/gsl_multifit_nlin.h>
#include <gsl/gsl_sort.h>
#include <gsl/gsl_statistics_double.h>
#include <gsl/gsl_integration.h>
#include <gsl/gsl_errno.h>
```

This code is used in chunk 17.

4.2 the beta function

return the logarithm of the (incomplete) beta function $B(x,a,b) = \int_0^x t^{a-1}(1-t)^{b-1}dt$ for $0 < x \leq 1$ and $a, b > 0$ using the Gauss hypergeometric function $B(x,a,b) = a^{-1}x^a(1-x)^b F(a+b,1,a+1,x)$

6 ⟨ beta function 6 ⟩ ≡

```
static long double betafunc(const double x, const double a, const double b)
{    /*
       the GSL incomplete beta function is normalised by the complete beta
       function */
  assert(x > 0);
  assert(a > 0);
  assert(b > 0);    /*
       the standard way would be gsl_sf_beta_inc(a, b, x) * gsl_sf_beta(a, b) */    /*
       return the logarithm of the beta function as log Γ(a) + log Γ(b) − log Γ(a + b) */
  const long double f = (long double)(x < 1 ? gsl_sf_hyperg_2F1 (a + b, 1, a + 1, x) : 1);
  assert(f > 0);    /*
       return 𝟙_{x<1}(log f + a log x + b log(1−x) − log a) + 𝟙_{x=1}(log Γ(a) + log Γ(b) − log Γ(a+b))
       */
  return (x < 1 ? (logl(f) + ((long double)((a * log(x)) + (b * log(1 − x)) − log(a)))) :
      (lgammal((long double) a) + lgammal((long double) b) − lgammal((long
      double)(a + b))));
}
```

This code is used in chunk 17.

4.3 binomial constant

compute the log of the binomial constant $\binom{n}{k}$

7 ⟨ binomial 7 ⟩ ≡

```
static long double binom(const int n, const int k)
{
  assert(k ≤ n);
  assert(k ≥ 0);
  assert(n ≥ 0);      /*
      log (n k) = log Γ(n + 1) − log Γ(k + 1) − log Γ(n − k + 1) */
  return (lgammal((long double)(n+1)) − lgammal((long double)(k+1)) − lgammal((long
      double)(n − k + 1)));
}
```

This code is used in chunk 17.

4.4 total beta rate

return the total beta rate $\binom{n}{k} B(\gamma, k-\alpha, n+\alpha-k)$.

8 ⟨ betarate 8 ⟩ ≡

```
static double lbetank(const int n, const int k, const double a, const double g)
{    /*
       return exp(log (n k) + log B(γ, k − α, n − k + α)) */    /*
       § 4.3 and § 4.2 */
  return ((double) expl(binom(n, k) + betafunc(g, ((double) k) − a, ((double)(n − k)) + a)));
}
```

This code is used in chunk 17.

4.5 the constant C in (4a)

compute the constant C in (4a) where we approximate $m_\infty \approx (2+(1+2^{1-\kappa})/(\kappa-1))/2$

9 ⟨ constantC 9 ⟩ ≡

```
static double constantC(const double a, const double k, const double c, const double
    cprime, const double g)
{   /*
    a = α, k = κ, c = c, cprime = c′ */   /*
    g = γ = 𝟙_{ζ(N)/N→K} K/(m_∞+K) + 𝟙_{ζ(N)/N→∞} and m_∞/(K+m_∞) = 1 − γ */
  const double m = (2. + (1. + pow(2, 1 − k))/(k − 1))/2.;   /*
    (4d); first taking κ = 2 */
  const double Ck = 1/pow(m, 2.);   /*
    c′_α = 𝟙_{α=1} c′ + 𝟙_{α∈(0,2)\{1}} c */
  const double caprime = (a > 1 ? c : (a < 1 ? c : cprime));   /*
    the integral in (4c) ∫_{𝟙_{ζ(N)/N→K} m_∞/(K+m_∞)}^1 (1 − y)^{1−α} y^{α−1} dy = ∫_0^1 𝟙_{0<t≤γ} t^{1−α}(1 − t)^{α−1} dt
    */   /*
    the incomplete beta function is normalised by the complete beta function in GSL */
  const double tegur = (g < 1 ? (gsl_sf_beta_inc(2 − a, a, g) * gsl_sf_beta(2 − a,
    a)) : gsl_sf_beta(2 − a, a));
  return (Ck + (a * caprime * tegur/pow(m, a)));
}
```

This code is used in chunk 17.

4.6 $\lambda_{n,k}$ (4a)

compute the transition rate $\lambda_{n,k}$ in (4a)

10 ⟨ lambdank 10 ⟩ ≡

```
static double lambdank(const int n, const int k, const double a, const double
        kappa, const double c, const double cprime, const double gm)
{
  assert(k > 1);
  assert(n > 1);
  assert(k ≤ n);      /*
      we approximate m_∞ ≈ (1/2)(2 + (1+2^{1−κ})/(κ−1)) */
  const double m = (2. + (1. + pow(2, 1 − kappa))/(kappa − 1))/2.;      /*
      taking κ = 2 so that C_κ = 2m_∞^{−2} (4d) */
  const double Ck = 2/pow(m, 2.);
  const double caprime = (a ≠ 1 ? c : cprime);      /*
      return (4a) using lbetank § 4.4 and constantC § 4.5 */
  return (((k < 3 ? (Ck ∗ ((double)(k ∗ (k − 1)))/2.) : 0) + (a ∗ caprime ∗ lbetank(n, k, a,
      gm)/pow(m, a)))/constantC(a, kappa, c, cprime, gm));
}
```

This code is used in chunk 17.

4.7 check the functions

check the functions in § 4.2, § 4.3, 4.4

11 ⟨ check 11 ⟩ ≡

```
static void checking(const double ix, const int ib)
{
  for (int k = 3; k ≤ ib; ++k) {
    printf("%f␣%d␣%d\n", ix, ib, k);
    printf("%Lf␣\n", betafunc(ix, (double) ib, (double) k));
    printf("%g\n", log(gsl_sf_beta_inc((double) ib, (double) k,
        ix) ∗ gsl_sf_beta((double) ib, (double) k)));
    printf("%Lf␣%g␣%g\n", binom(ib, k), log((double) gsl_sf_choose((unsigned) ib,
        (unsigned) k)), gsl_sf_lnchoose((unsigned int) ib, (unsigned int) k));
  }
}
```

This code is used in chunk 17.

4.8 the jump rate

compute the jump rate of going from i to j blocks $\lambda_{i,i-k+1}$ with $\lambda_{n,k}$ the rate of merging k blocks; then $k = n - j + 1$

12 ⟨ jumprate 12 ⟩ ≡

```
static double qij(const int i, const int j, const double a, const double kpp, const
        double c, const double cpr, const double g)
{   /*
      kpp is κ ≥ 2 */   /*
      cpr is c'_α = c when α ≠ 1, otherwise c' ∈ (0, 1) when α = 1 and κ > 2 */   /*
      g is γ (4b) */
  assert(i > 1);
  assert(j > 0);
  assert(j < i);   /*
      i → j = i − k + 1 so k = i − j + 1 ; using lambdank § 4.6 */
  return (lambdank(i, i − j + 1, a, kpp, c, cpr, g));
}
```

This code is used in chunk 17.

4.9 QP

compute the matrices Q and P

13 ⟨ QP 13 ⟩ ≡

```
static void QP(const int n, const double a, const double kpp, const double cconst, const
        double cprime, const double g, gsl_matrix * Q, gsl_matrix * P)
{   /*
      compute matrices Q qij and P pij */   /*
      a is α ∈ (0, 2) */   /*
      kpp is κ ≥ 2 */   /*
      cconst is c > 0 */   /*
      cprime is c′ ∈ (0, 1) */   /*
      g is γ (4b) */
  int i, j;
  double s = 0;
  double x = 0;
  for (i = 2; i ≤ n; i++) {
    assert(i ≤ n);
    s = 0.;
    for (j = 1; j < i; j++) {   /*
          using qij from § 4.8 */
      x = qij(i, j, a, kpp, cconst, cprime, g);
      s += x;
      gsl_matrix_set(Q, i, j, x);
      gsl_matrix_set(P, i, j, x);
    }
    assert(s > 0);
    gsl_matrix_set(Q, i, i, s);
    for (j = 1; j < i; j++) {
      assert(j < i);
      gsl_matrix_set(P, i, j, gsl_matrix_get(P, i, j)/s);
    }
  }
}
```

This code is used in chunk 17.

4.10 G

compute the G matrix

14 ⟨ Gmatrix 14 ⟩ ≡

```
static void gmatrix(const int n, gsl_matrix * G, gsl_matrix * Q, gsl_matrix * P)
{
  int i, k, m;
  double s = 0.0;      /*
      initialise the diagonal */
  for (i = 2; i ≤ n; i++) {
    assert(gsl_matrix_get(Q, i, i) > 0);
    gsl_matrix_set(G, i, i, 1./gsl_matrix_get(Q, i, i));
  }
  for (i = 3; i ≤ n; i++) {
    assert(i ≤ n);
    for (m = 2; m < i; m++) {
      s = 0.;
      for (k = m; k < i; k++) {
        assert(i ≤ n);
        assert(k ≤ n);
        assert(m ≤ n);
        s += gsl_matrix_get(P, i, k) * gsl_matrix_get(G, k, m);
      }
      gsl_matrix_set(G, i, m, s);
    }
  }
}
```

This code is used in chunk 17.

4.11 the matrix $p^{(n)}[k,b]$

compute the matrix $p^{(n)}[k,b]$

15 ⟨ pnkb 15 ⟩ ≡

```
static void pnb(int n, gsl_matrix * lkb, gsl_matrix * G, gsl_matrix * P)
{     /*
        j is n' nprime from [BBE2013a, Prop A1] */     /*
        lnb is the matrix p^(nprime)[k,b]; used for each fixed k */
  gsl_matrix * lnb = gsl_matrix_calloc(n + 1, n + 1);
  int k, b, j, i;
  double s = 0.0;
  gsl_matrix_set(lkb, n, 1, 1.0);
  for (k = 2; k < n; k++) {
    for (i = k; i ≤ n; i++) {
      for (b = 1; b ≤ i − k + 1; b++) {
        gsl_matrix_set(lnb, i, b, (k ≡ i ? (b ≡ 1 ? 1.0 : 0.0) : 0.0));
        s = 0.;
        for (j = k; j < i; j++) {
          gsl_matrix_set(lnb, i, b, gsl_matrix_get(lnb, i,
              b) + (b > i − j ? (((double)(b − i + j)) * gsl_matrix_get(lnb, j,
              b−i+j)*(gsl_matrix_get(P, i, j)*gsl_matrix_get(G, j, k)/gsl_matrix_get(G,
              i, k))/((double) j)) : 0.0) + (b < j ? ((((double)(j − b)) * gsl_matrix_get(lnb,
              j, b) * (gsl_matrix_get(P, i, j) * gsl_matrix_get(G, j, k)/gsl_matrix_get(G, i,
              k))/((double) j))) : 0.0));
        }
      }
    }
    for (j = 1; j ≤ (n − k + 1); j++) {
      gsl_matrix_set(lkb, k, j, gsl_matrix_get(lnb, n, j));
    }
    gsl_matrix_set_zero(lnb);
  }
  gsl_matrix_free(lnb);
}
```

This code is used in chunk 17.

4.12 compute the expected spectrum

compute the expected spectrum $\mathbb{E}\left[L_i(n)\right]/\mathbb{E}\left[L(n)\right]$

16 ⟨ ebi 16 ⟩ ≡

```
static void ebi(const int n, const double a, const double kappa, const double
        cconst, const double cprime, const double gm)
{   /*
      n is sample size */   /*
      a is α ∈ (0, 2) */   /*
      kappa is κ ≥ 2 */   /*
      cconst is c > 0 */   /*
      cprime is c′ ∈ (0, 1) */   /*
      gm is γ (4b) */
  gsl_matrix * P = gsl_matrix_calloc(n + 1, n + 1);
  gsl_matrix * Q = gsl_matrix_calloc(n + 1, n + 1);
  gsl_matrix * G = gsl_matrix_calloc(n + 1, n + 1);
  gsl_matrix * Pn = gsl_matrix_calloc(n + 1, n + 1);   /*
      iA is κ */
  const double iA = kappa;
  const double minfty = (2. + (1. + pow(2., 1. − iA))/(iA − 1.))/2.;   /*
      compute the Q and P matrices § 4.9 */
  QP(n, a, iA, cconst, cprime, gm, Q, P);   /*
      the G matrix § 4.10 */
  gmatrix(n, G, Q, P);   /*
      the p^(n)[k, b] matrix § 4.11 */
  pnb(n, Pn, G, P);
  int b, k;
  double s = 0.0;   /*
      eb will store E[L(n)] */
  double eb = 0.0;
  double *ebi = (double *) calloc(n, sizeof(double));
  for (b = 1; b < n; b++) {
    s = 0.;
    for (k = 2; k ≤ n − b + 1; k++) {
      s = s + (gsl_matrix_get(Pn, k, b) * ((double) k) * gsl_matrix_get(G, n, k));
    }
    ebi[b] = s;
    eb += s;
  }
  assert(eb > 0);
  double x;
  for (b = 1; b < n; ++b) {
    x = ((double) b)/((double) n);
    printf("%g,%g\n", log(x) − log(1 − x), log(ebi[b]/eb) − log(1.0 − (ebi[b]/eb)));
  }
  gsl_matrix_free(P);
  gsl_matrix_free(Q);
  gsl_matrix_free(G);
  gsl_matrix_free(Pn);
  free(ebi);
}
```

This code is used in chunk 17.

4.13 the main function

17

```
/*
§ 4.1 */
⟨ includes 5 ⟩ /*
§ 4.2 */
⟨ beta function 6 ⟩ /*
§ 4.3 */
⟨ binomial 7 ⟩ /*
§ 4.4 */
⟨ betarate 8 ⟩ /*
§ 4.5 */
⟨ constantC 9 ⟩ /*
§ 4.6 */
⟨ lambdank 10 ⟩ /*
§ 4.7 */
⟨ check 11 ⟩ /*
§ 4.8 */
⟨ jumprate 12 ⟩ /*
§ 4.9 */
⟨ QP 13 ⟩ /*
§ 4.10 */
⟨ Gmatrix 14 ⟩ /*
§ 4.11 */
⟨ pnkb 15 ⟩ /*
§ 4.12 */
⟨ ebi 16 ⟩
int main(int argc, char *argv[])
{ /*
ebi § 4.12 */ /*
ebi( const int n, const double a, const double kappa, const double cconst, const
double cprime, const double gm ); */ /* ebi( atoi(argv[1]), atof(argv[2]),
atof(argv[3]), atof(argv[4]), atof(argv[5]), atof(argv[6]) ); */
ebi(100, 0.01, 2, 1, 1, 1);
return GSL_SUCCESS;
}
```

5 conclusions and bibliography

For $i = 1, 2, \ldots, n-1$ where n is sample size we compute

$$\mathbb{E}\left[L_i(n)\right] = \sum_{k=2}^{n-i+1} k p^{(n)}[k,i] g(n,k) \tag{5}$$

by a recursion; $g(n,m) = \mathbb{E}\left[\int_0^\infty \mathbb{1}_{\{\#\xi^n(s)=m\}} ds : \xi^{(n)}(0) = n\right]$ is the expected amount time $\{\xi^n\}$ has m blocks when starting with n, and $p^{(n)}[k,b]$ is the probability a given branch when there are k branches will support exactly b leaves; see [BBE2013a] for details. Equation (5) can be used to compute $\mathbb{E}\left[L_i(n)\right]$ for any coalescent provided $p^{(n)}[k,i]g(n,k)$ can be computed; [Blath2016] apply (5) to Ξ-coalescents admitting simultaneous multiple mergers. On the implementation side, $p^{(n)}[k,i]$ involves nested loops.

Figure 1 records an example

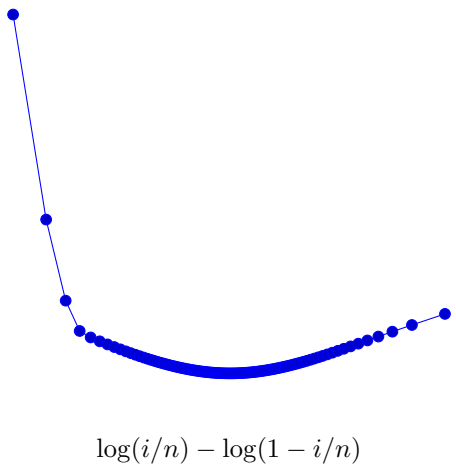

Figure 1: *An example approximation of* $\mathbb{E}\left[R_i(n)\right]$ *graphed as logits against* $\log(i/n) - log(1-i/n)$ *where* n *is sample size*

References

[Blath2016] Jochen Blath and Mathias Christensen Cronjäger and Bjarki Eldon and Matthias Hammer. Theoretical Population Biology 36–50, The site-frequency spectrum associated with Ξ-coalescents, https://dx.doi.org/10.1016/j.tpb.2016.04.002, 110, 2016.

[BBE2013a] M Birkner and J Blath and B Eldon. Genetics, 1037–1053. Statistical properties of the site-frequency spectrum associated with Λ-coalescents. 195, 2013.

[Eld24] Bjarki Eldon. Gene genealogies in haploid populations evolving according to sweepstakes reproduction. In preparation, 2024+.

[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.

[Sch03] J Schweinsberg. Coalescent processes obtained from supercritical Galton-Watson processes. *Stoch Proc Appl*, 106:107–139, 2003.

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

Index

List of Refinements

⟨ Gmatrix 14 ⟩ Used in chunk 17.
⟨ QP 13 ⟩ Used in chunk 17.
⟨ beta function 6 ⟩ Used in chunk 17.
⟨ betarate 8 ⟩ Used in chunk 17.
⟨ binomial 7 ⟩ Used in chunk 17.
⟨ check 11 ⟩ Used in chunk 17.
⟨ constantC 9 ⟩ Used in chunk 17.
⟨ ebi 16 ⟩ Used in chunk 17.
⟨ includes 5 ⟩ Used in chunk 17.
⟨ jumprate 12 ⟩ Used in chunk 17.
⟨ lambdank 10 ⟩ Used in chunk 17.
⟨ pnkb 15 ⟩ Used in chunk 17.