Beta coalescents when sample size is large
— estimating $\mathbb{E}[R_i(n)]$ for the Beta$(\gamma, 2-\alpha, \alpha)$-coalescent

Bjarki Eldon[1] [2]

Let $\{\xi^n(t) : t \geq 0\}$ be the Beta$(\gamma, 2-\alpha, \alpha)$-coalescent, $\#A$ is the cardinality of a given set A, n sample size, $L_i^N(n) \equiv \int_0^{\tau(n)} \# \{\xi \in \xi^n(t) : \#\xi = i\}\, dt$ and $L(n) \equiv \int_0^{\tau(n)} \#\xi^n(t) dt$ and $\tau(n) \equiv \inf\{t \geq 0 : \#\xi^n(t) = 1\}$ for $i \in \{1, 2, \ldots, n-1\}$; $R_i(n) \equiv L_i(n) / \sum_j L_j(n)$ for $i = 1, 2, \ldots, n-1$. Then $L_i^N(n)$ is interpreted as the random total length of branches supporting $i \in \{1, 2, \ldots, n-1\}$ leaves, with the length measured in coalescent time units, and n sample size. We then have $L(n) = L_1(n) + \cdots + L_{n-1}(n)$. With this C++ code one estimates the functionals $\mathbb{E}[R_i(n)]$ of gene genealogies described by the Beta$(\gamma, 2-\alpha, \alpha)$-coalescent where $0 < \gamma \leq 1$ and $1 \leq \alpha < 2$. The Beta$(\gamma, 2-\alpha, \alpha)$-coalescents are a family of Λ-coalescents [Pit99, DK99, Sag99]; the transition rates are

$$\lambda_{n,k} = \binom{n}{k} \frac{B(\gamma, k-\alpha, n-k+\alpha)}{B(\gamma, 2-\alpha, \alpha)}$$

for $k = 2, 3, \ldots, n$ where $B(x, a, b) = \int_0^x t^{a-1}(1-t)^{b-1} dt$ for $0 < x \leq 1$ and $a, b > 0$ [CDEH25]. The Beta$(\gamma, 2-\alpha, \alpha)$-coalescent extends the Beta$(2-\alpha, \alpha)$-coalescent derived from a model of sweepstakes reproduction (skewed offspring number distribution) [Sch03].

Contents

[1] beldon11@gmail.com

[2] Supported by Deutsche Forschungsgemeinschaft (DFG) - Projektnummer 273887127 through DFG SPP 1819: Rapid Evolutionary Adaptation grant STE 325/17 to Wolfgang Stephan; acknowledge funding by the Icelandic Centre of Research (Rannís) through an Icelandic Research Fund (Rannsóknasjóður) Grant of Excellence no. 185151-051 to Einar Árnason, Katrín Halldórsdóttir, Alison M. Etheridge, Wolfgang Stephan, and BE; Start-up module grants through SPP 1819 with Jere Koskela and Maite Wilke-Berenguer, and with Iulia Dahmer.
December 29, 2025

1 Copyright

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.

Compiles on `Linux Debian trixie/sid` with kernel `6.12.10-amd64` and `ctangle 4.11` and `g++ 14.2` and `GSL 2.8`

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
      g++ -Wall -Wextra -pedantic -std=c++26 -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 <program call>`

Use `cppcheck` to check the code

`cppchek --enable=all --language=c++ <prefix>.c`

To generate estimates on a computer with several CPUs it may be convenient to put in a text file (`simfile`):

`./a.out $(shuf -i 484433-83230401 -n1) > resout<i>`

for $i = 1, \ldots, y$ and use `parallel`[Tan11]

`parallel --gnu -jy ::::  ./simfile`

3 introduction

Let $\{\xi^n(t) : t \geq 0\}$ be the Beta$(\gamma, 2-\alpha, \alpha)$-coalescent, $\#A$ is the cardinality of a given set A, n sample size, $L_i^N(n) \equiv \int_0^{\tau(n)} \# \{\xi \in \xi^n(t) : \#\xi = i\} \, dt$ and $L(n) \equiv \int_0^{\tau(n)} \#\xi^n(t)dt$ and $\tau(n) \equiv \inf \{t \geq 0 : \#\xi^n(t) = 1\}$ for $i \in \{1, 2, \ldots, n-1\}$; $R_i(n) \equiv L_i(n)/\sum_j L_j(n)$ for $i = 1, 2, \ldots, n-1$. Then $L_i^N(n)$ is interpreted as the random total length of branches supporting $i \in \{1, 2, \ldots, n-1\}$ leaves, with the length measured in coalescent time units, and n sample size. We then have $L(n) = L_1(n) + \cdots + L_{n-1}(n)$.

We estimate $\mathbb{E}[R_i(n)]$ when the gene genealogy is determined by the Beta$(\gamma, 2-\alpha, \alpha)$-coalescent with transition rates

$$\lambda_{n,k} = \binom{n}{k} \frac{B(\gamma, k-\alpha, n-k+\alpha)}{B(\gamma, 2-\alpha, \alpha)} \tag{1}$$

for $2 \leq k \leq n$ and $B(\gamma, a, b) = \int_0^\gamma t^{a-1}(1-t)^{b-1}dt$ and $a, b > 0$ and $0 < \gamma \leq 1$.

The Beta$(\gamma, 2-\alpha, \alpha)$-coalescent can be shown to describe the random gene genealogies of a sample when the sample comes from a haploid panmictic population of constant size evolving according to randomly increased recruitment and when there is an upper bound on the random number of potential offspring any arbitrary individual can produce. The upper bound translates to the parameter γ of the Beta$(\gamma, 2-\alpha, \alpha)$-coalescent [CDEH25].

The algorithm is summarised in § 4, the code follows in § 4.1–§ 4.12; we conclude in § 5. Comments within the code are in **this font and colour**

4 Code

Write $[n] = \{1, 2, \ldots, n\}$ for any natural number n. Let n denote the sample size and $m \in [n]$ the current number of blocks. We are interested in the branch lengths and require the block sizes $(b_1, \ldots, b_m)$ where $b_j \in [n]$ and $b_1 + \cdots + b_m = n$ are the current block sizes

1. $(r_1(n), \ldots, r_{n-1}(n)) \leftarrow (0, \ldots, 0)$
2. for each of M experiments; § 4.11:
   (a) set $(b_1, \ldots, b_n) \leftarrow (1, \ldots, 1)$.
   (b) set current branch lengths $\ell_i(n) \leftarrow 0$ for $i \in [n-1]$
   (c) set the current number of blocks $m \leftarrow n$
   (d) **while** $m > 1$ (at least two blocks; see § 4.10) :
      i. sample exponential time t with rate $\lambda_{m,2} + \cdots + \lambda_{m,m}$ (1)
      ii. update the current branch lengths $\ell_b(n) \leftarrow t + \ell_b(n)$ for $b = b_1, \ldots, b_m$; § 4.8
      iii. sample merger size $j \in \{2, 3, \ldots, m\}$ § 4.7
      iv. given merger size shuffle the blocks and merge j of them § 4.10
      v. update current number of blocks $m \leftarrow m - j + 1$
   (e) given a realisation $\ell_1(n), \ldots, \ell_{n-1}(n)$ of branch lengths update the estimate of $\mathbb{E}[R_i(n)]$; $r_i(n) \leftarrow r_i(n) + \ell_i(n) / \sum_j \ell_j(n)$ § 4.9
3. return an estimate $(1/M) r_i(n)$ of $\mathbb{E}[R_i(n)]$ for $i = 1, 2, \ldots, n-1$

4.1 includes

the included libraries; we use the `GSL` and `boost` libraries

5 ⟨includes 5⟩≡

```
#include <iostream>
#include <fstream>
#include <vector>
#include <random>
#include <functional>
#include <memory>
#include <utility>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <cmath>
#include <list>
#include <string>
#include <fstream>
#include <chrono>
#include <forward_list>
#include <assert.h>
#include <math.h>
#include <unistd.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <gsl/gsl_sf.h>
#include <boost/math/special_functions/beta.hpp>
```

This code is used in chunk 16.

4.2 the random number generator

initialise the random number engines; we will not go into discussions about how to get a computer to give us a "random" number

6 ⟨gslrng 6⟩ ≡

```
/* the GSL random number engine */
gsl_rng *rngtype;
/* obtain a seed out of thin air for the STL random number engine */
std::random_device randomseed;
/* The STL standard Mersenne twister random number engine seeded with randomseed() */
std::mt19937_64 rng(randomseed());
/* set up and initialise the GSL random number generator */
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 16.

4.3 incomplete beta function

use the GSL Gauss hypergeometric function to compute the incomplete beta function; we have the representations

$$B(x,a,b) = x^a F(a, 1-b; a+1; x)/a$$
$$B(x,a,b) = x^a (1-x)^b F(a+b, 1; a+1; x)/a$$

return the logarithm of $B(x,a,b)$

7 ⟨ log of incomplete Beta 7 ⟩ ≡

```
static long double lnincbetaGF(const long double &a, const long double &b, const
        long double &x)
{
  return logl(static_cast<long double>(gsl_sf_hyperg_2F1(a + b, 1, a + 1,
      x))) + (a * logl(x)) + (b * log(1. - x)) - log(a);
}
```

This code is used in chunk 16.

4.4 the incomplete beta function using boost

the incomplete beta function using the boost library

8 ⟨incomplete beta using boost 8⟩ ≡

```
static double incbeta(const double &a, const double &b, const double &x)
{
  assert(x <= 1.);
  assert(0 <= x);    /*
        if using the GSL library gsl_sf_beta_inc(a, b, x) * gsl_sf_beta(a, b)   */
  return (x < 1 ? boost::math::beta(static_cast<long double>(a), static_cast<long
      double>(b), static_cast<long double>(x)) : gsl_sf_beta(a, b));
}
```

This code is used in chunk 16.

4.5 the merger rate

compute the merger rate $\binom{n}{k}B(\gamma, k-\alpha, n-k+\alpha)/B(\gamma, 2-\alpha, \alpha)$

9 ⟨ merger rate 9 ⟩ ≡

```
static double rate(const double &m, const double &k, const double &a, const long
        double &x)
{
  assert(k − a > 0);
  assert(k ≤ m);
  assert(m + a − k > 0);      /*
       using incbeta from § 4.4   */
  return static_cast⟨double⟩(expl(lgammal(m + 1) − lgammal(k + 1) − lgammal(m − k +
      1) + logl(incbeta(k − a, m + a − k, x)) − logl(incbeta(2 − a, a, x))));
}
```

This code is used in chunk 16.

4.6 the total merger rate

compute the total merger rate $\lambda_n = \lambda_{n,2} + \cdots + \lambda_{n,n}$

10 ⟨lambdan 10⟩ ≡

```
static void totalrate (const double &n, const double &a, const double &x, std::vector
    < double > &v )
{
  for (double m = 2; m <= n; ++m) {
    for (double j = 2; j <= m; ++j) {
      assert(j <= m);      /*
            using rate from § 4.5 */
      v[m] += rate(m, j, a, x);
    }
  }
}
```

This code is used in chunk 16.

4.7 sample merger size

sample merger size using the transition rates, returning $\min\{j : U \leq \sum_{i=2}^{j} \lambda_{n,i}/\lambda_n\}$ where U is a random uniform from the unit interval and $\lambda_n \equiv \lambda_{n,2} + \cdots + \lambda_{n,n}$ (1)

11 $\langle$ mergersize 11 $\rangle \equiv$

```
static double getmerger (const double &m, const double &a, const double &x, const
    std::vector < double > &v )
{     /*
        m is the current number of lines; a is α; x is γ;   */      /*
        v stores the λ_n values */      /*
        sample a random uniform */
  const double u = gsl_rng_uniform(rngtype);
  double j = 2;      /*
        rate from § 4.5 */
  double s = rate(m, j, a, x);
  while (u > s/v[static_cast<int>(m)]) {
    ++j;
    assert(j <= m);
    s += rate(m, j, a, x);
  }
  return j;
}
```

This code is used in chunk 16.

4.8 update branch lengths $L_i(n)$

update the branch lengths $L_i(n)$ from a current configuration of block sizes

12 ⟨updateLin 12⟩ ≡

```
static void updateb (const double &timi, const std::vector < int > &tre, std::vector <
    double > &b ) {      /*
    timi is the sampled waiting time in the configuration given in tre  */
assert(timi > 0);
std::for_each (tre.begin( ), tre.end( ), [&timi, &b](const int t)
{
  assert(t > 0);
  b[0] += timi;
  b[t] += timi;
}
) ; }
```

This code is used in chunk 16.

4.9 update estimate of $\mathbb{E}[R_i(n)]$

update the estimate of $\mathbb{E}[R_i(n)]$ for $i = 1, 2, \ldots, n-1$

13 ⟨ updateri 13 ⟩ ≡

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

This code is used in chunk 16.

4.10 one tree

generate one realisation of $L_i(n)$ for $i = 1, 2, \ldots, n-1$

14 ⟨generate one tree 14⟩ ≡

```
static void genealogy (const int &n, const double &a, const double &x, const std::vector
    < double > &v, std::vector < double > &vri ) {
std::vector < int > t(n, 1);
std::size_t ms
{ }
;
double timi
{ }
;
int newb
{ }
; std::vector < double > vb(n);
std::size_t q
{ }
;
while (t.size( ) > 1) {     /*
        sample waiting time until next merger */
  timi = gsl_ran_exponential(rngtype, 1./v[t.size( )]);
  assert(timi > 0);     /*
        update branch lengths § 4.8 */
  updateb(timi, t, vb);     /*
        get the size of next merger § 4.7 */
  ms = static_cast⟨std::size_t⟩(getmerger(static_cast⟨double⟩(t.size( )), a, x, v));     /*
        shuffle the blocks and merge the rightmost ms blocks */
  std::shuffle(t.begin( ), t.end( ), rng);     /*
        get the size of the new block */
  newb = std::accumulate(t.rbegin( ), t.rbegin( ) + ms, 0);     /*
        q is the current number of blocks */
  q = t.size( );     /*
        remove the merged blocks */
  t.resize(q − ms);     /*
        add the new block newb to the configuration */
  t.push_back(newb);
}     /*
        given realised branch lengths update the estimate of E[R_i(n)] § 4.9 */
updateri(vb, vri); }
```

This code is used in chunk 16.

4.11 estimate $\mathbb{E}[R_i(n)]$

15 ⟨get an estimate of $\mathbb{E}[R_i(n)]$ 15⟩ ≡

```
static void estimate(const double &n, const double &a, const double &K) {    /*
        approximate the mean mu = m∞ ≈ (2 + (1 + 2^(1−α))/(α − 1))/2 */    /*
        need α > 1 when applying a cutoff; see the approximation of m∞ below */
    const double mu = (a > 1 ? ((1 + (pow(2., 1. − a)/(a − 1))) + (1 + (1/(a − 1))))/2. : 0);
      /*
       K is the cutoff constant; K = 0 is taken as unbounded distribution of number of
       potential offspring translating to complete Beta(2 − α, α)-coalescent; otherwise
       the cutoff is K/(m∞ + K) */    /*
        if α = 1 taking the cutoff as K */
    const double p = (a > 1 ? (K > 0 ? K/(mu + K) : 1) : K);
    std::vector < double > v(static_cast⟨int⟩(n) + 1);    /*
        totalrate § 4.6 */
    totalrate(n, a, p, v); std::vector < double > vri(static_cast⟨int⟩(n));    /*
        set to 10^5 number of experiments */
    int r = 1 · 10^5 + 1;
    while (−−r > 0) {    /*
          genealogy § 4.10 */
      genealogy(static_cast⟨int⟩(n), a, p, v, vri);
    }    /*
          print the estimates of E[R_i(n)] summer over the experiments */
    std::for_each (vri.begin(), vri.end(), [](const auto &x)
    {
      std::cout ≪ x ≪ '\n';
    }
    ); }
```

This code is used in chunk 16.

4.12 the main module

The *main* function

16

```
/*
§ 4.1 */
⟨ includes 5 ⟩ /*
    § 4.2 */
  ⟨ gslrng 6 ⟩ /*
    § 4.3 */
  ⟨ log of incomplete Beta 7 ⟩ /*
    § 4.4 */
  ⟨ incomplete beta using boost 8 ⟩ /*
    § 4.5 */
  ⟨ merger rate 9 ⟩ /*
    § 4.6 */
  ⟨ lambdan 10 ⟩ /*
    § 4.7 */
  ⟨ mergersize 11 ⟩ /*
    § 4.8 */
  ⟨ updateLin 12 ⟩ /*
    § 4.9 */
  ⟨ updateri 13 ⟩ /*
    § 4.10 */
  ⟨ generate one tree 14 ⟩ /*
    § 4.11 */
  ⟨ get an estimate of E[R_i(n)] 15 ⟩
  int main(int argc, char *argv[])
  { /*
      initialise the GSL random number generator rngtype using setup_rng § 4.2 */
    setup_rng(static_cast⟨unsigned long int⟩(atoi(argv[1]))); /*
      estimate E[R_i(n)] using estimate § 4.11 */
    estimate(atof(argv[1]), atof(argv[2]), atof(argv[3]));
    gsl_rng_free(rngtype);
    return GSL_SUCCESS;
  }
```

5 conclusion and bibliography

The Beta($\gamma, 2-\alpha, \alpha$)-coalescent [CDEH25] extends the Beta($2-\alpha, \alpha$)-coalescent of [Sch03]; when $\gamma = 1$ one recovers the Beta($2-\alpha, \alpha$)-coalescent of [Sch03]. Moreover, the upper bound affects the predicted site-frequency spectrum; for any $1 < \alpha < 2$ the spectrum can be indistinguishable from the one predicted by the Kingman-coalescent provided γ is small enough [CDEH25]. The Beta($\gamma, 2-\alpha, \alpha$)-coalescent is determined by $\gamma \in (0, 1]$ and $\alpha \in (1, 2)$ and so one should jointly estimate the two parameters. The Beta($\gamma, 2-\alpha, \alpha$)-coalescent would be suitable to compare against population genetic data inherited in a haploid manner, e.g. the site-frequency spectrum of the mtDNA of diploid populations.

References

[CDEH25] JA Chetwyn-Diggle, Bjarki Eldon 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.

[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.

[Sag99] S Sagitov. The general coalescent with asynchronous mergers of ancestral lines. *J Appl Probab*, 36:1116–1125, 1999.

[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