Simulating Beta-coalescents

Bjarki Eldon[1]
Leibniz Institute for Evolution and Biodiversity Science
Museum für Naturkunde
10115 Berlin, Germany
August 31, 2021

Abstract

We describe C++ code for simulating a particular class of Λ-coalescents referred to as Beta-coalescents, adapting the algorithm in[(2)]. The incomplete Beta-coalescent is derived in[(1)], and is a variant of the complete Beta-coalescent[(3)].

1 License

2 Background

A Λ-coalescent is a continuous-time Markov chain on the partitions of $\mathbb{N} := \{1, 2, \ldots\}$; restricted to $\{1, \ldots, n\}$ for some $n \in \mathbb{N}$ it has rates, for $2 \leq k \leq n$,

$$\lambda_{n,k} = \int_0^1 t^{k-2}(1-t)^{n-k}\Lambda(dt) + a\mathbb{1}_{\{k=2\}} \tag{1}$$

[1]Supported by Deutsche Forschungsgemeinschaft (DFG) - Projektnummer 273887127 through DFG SPP 1819: Rapid Evolutionary Adaptation grant STE 325/17-2 to Wolfgang Stephan; Icelandic Centre of Research through an Icelandic Research Fund Grant of Excellence no. 185151-051 to Einar Árnason, Katrín Halldórsdóttir, Wolfgang Stephan, Alison Etheridge, and BE.

where Λ is a finite measure on $(0,1]$, i.e. without an atom at zero. A Beta-coalescent is a Λ-coalescent with rate, for $1 \leq \alpha < 2$,

$$\lambda_{n,k}^{\text{Beta}} = \frac{1}{B(M; 2-\alpha, \alpha)} \int_0^M t^{k-\alpha-1}(1-t)^{n+\alpha-k-1} dt \tag{2}$$

where $B(M; a, b)$ is the (incomplete) beta-function $B(M; a, b) := \int_0^M t^{a-1}(1-t)^{b-1} dt$ for a given constant $M \in (0, 1]$.

An obvious approach to sample the process would be to compute the total rate of merging k blocks,

$$\mu_{n,k}^{\text{Beta}} := \binom{n}{k} \lambda_{n,k}^{\text{Beta}} \tag{3}$$

for all $k \in \{2, \ldots, m\}$ and for all $2 \leq m \leq n$, with n the given sample size, and use the resulting probability distribution, sampling a merger size as

$$\max \left\{ i \in \{2, \ldots, m\} : u < \frac{\mu_{m,i}^{\text{Beta}}}{\sum_{j=2}^{m} \mu_{m,j}^{\text{Beta}}} \right\} \tag{4}$$

where u is a random uniform on the unit interval. However, for large n it may not be feasible to numerically evaluate $\mu_{n,k}^{\text{Beta}}$.

The code returns estimates of $\mathbb{E}[B_i(n)/B(n)]$ where $B_i(n)$ is the random length of branches supporting $i \in \{1, \ldots, n-1\}$ leaves, and $B(n) := B_1(n) + \cdots + B_{n-1}(n)$ is the total tree length. The ratio $B_i(n)/B(n)$ is well defined since $B(n) > 0$ almost surely, and $0 \leq B_i(n)/B(n) \leq 1$ almost surely.

3 Compilation and call

Compile the document with

```
latex -shell-escape clambdabetalargen.tex
```

Compile the code with

```
c++ -Wall -Wextra -Wshadow -Wnon-virtual-dtor -m64 -std=c++20
-march=native -pedantic -DNDEBUG clambdabetalargen_minimal.cpp
-lm -lgsl -lgslcblas
```

The call is (see § 12 and § 13), given the executable is `a.out`:

```
./a.out <sample size> <alpha parameter> <K parameter>
$(shuf -i 1000-100000 -n1)
```

4 Includes

The included libraries.

```
#include <iostream>
#include <vector>
#include <random>
#include <functional>
#include <memory>
#include <utility>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <cmath>
#include <assert.h>
#include <math.h>
#include <unistd.h>
#include <omp.h>
```

5 Random number generators

```
// obtain a seed out of thin air for the random number engine
 std::random_device randomseed;
//  Standard mersenne twister  random number engine
// seeded with randomseed()
std::mt19937_64 rng(randomseed());

// the GSL random number generator
gsl_rng * rngtype;

// initialising 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 Sample from a beta-distribution

Use rejection sampling to sample from a beta-distribution Beta$(M; 2-\alpha, \alpha)$ in case of incomplete Beta-coalescent Eq (2).

```
static double xreject( const double a,  const double M)
{

 double x = gsl_ran_beta(rngtype, 2.0 - a, a) ;

 while( (x > M) || ( x <= 0.) ){
   x = gsl_ran_beta(rngtype, 2.0 - a, a) ; }

  assert( x > 0. );
  assert( x <= M);
  return( x);
}
```

7 Sample sojourn time

Sample time during which we see a given number of blocks, and a variate from a beta-distribution. Note the total rate of mergers in a Λ-coalescent without an atom at zero can be written

$$\lambda_n = \int_0^1 \mathbb{1}_{\{0<t\leq M\}} \left(1-(1-t)^n - nt(1-t)^{n-1}\right) t^{-2} \Lambda(dt). \tag{5}$$

```
static double sample_interval_time(const double n, const double a,
                                   const double M,  double* x)
{
  // n is current number of  blocks
  // a is alpha
  // M is upper limit on beta-function integral

double timi = 0. ;
const double R = n*(n - 1.)/2. ;

if ( a < 2. ){
do{
  timi = timi + ( -log(1. - gsl_rng_uniform(rngtype) )/R );
  x[0] = xreject( a,  M); }
```

```
  while ( gsl_rng_uniform(rngtype) >
  (((1. - pow(1.-x[0], n) - (n*x[0]*pow(1.-x[0], n-1.)))/(x[0]*x[0]))/R));
}
else{
   timi =  ( -log(1. - gsl_rng_uniform(rngtype) )/R);}

 return ( timi );
}
```

8 Sample merger size

Sample a merger size given a probability of participating in a merger computed in § 7.

```
static size_t samplek( const size_t n,  const double x )
{

 size_t k = 2 + (x > 0.0 ?  gsl_ran_binomial(rngtype, x, n - 2) : 0);

  while(gsl_rng_uniform(rngtype) > 2./static_cast<double>(k*(k - 1))){
    k = 2 + (x > 0.0 ? gsl_ran_binomial(rngtype, x, n - 2) : 0);}

  return(k);
}
```

9 Update branch lengths

Update branch lengths of the branch length spectrum $(B_1(n),\ldots,B_{n-1}(n))$ given a realised sojourn time from § 7.

```
static void updateBranchLengths(  std::vector<double>& i_b,
       const std::vector<size_t>& i_t, const double i_time )
{
 // i_t is current vector of block sizes
 // i_time is the sampled sojourn time
  for( const auto &y : i_t){
    // add branch length to total tree length
    i_b[0] += i_time ;
   // add to branch length y
    i_b[ y ] += i_time;}
}
```

10 Sum merging blocks

Sum the blocks sampled to merge given merger size sampled in § 8; returns the size of the new block. Given merger size k, the last k blocks of the shuffled tree (vector of block sizes) are merged.

```
static size_t sum_merging_blocks(const std::vector<size_t>& i_t,
                                 const size_t merger_size )
{

  size_t m = 0;
  for( size_t i = i_t.size() - merger_size; i < i_t.size(); ++i){
    m += i_t[i] ;}
  return(m);
}
```

11 Update estimate of $\mathbb{E}[B_i(n)/B(n)]$

Update estimate of $\mathbb{E}[B_i(n)/B(n)]$ given a realisation of branch lengths.

```
static void update_estimate_ebib(std::vector<double>& estimate_ebib,
           const std::vector<double>& vbi, const size_t sample_size)
{
// vbi is a vector of branch lengths
  for( size_t i = 1 ; i < sample_size ; ++i){
    estimate_ebib[i] += vbi[i]/vbi[0]; }
}
```

12 Estimate $\mathbb{E}[B_i(n)/B(n)]$

Estimate $\mathbb{E}[B_i(n)/B(n)]$ by sampling trees and realisations of branch lengths.

```
static void estimate_ebib(const size_t n,
                          const double param_alpha,
                          const double param_K)
{

// n is sample size;
// param_alpha is the alpha parameter for the
// distribution of juveniles;
```

```
// param_K is the cutoff parameter on the number of
// juveniles


  const int number_experiments = 100 ;
  int trials = number_experiments + 1;

  double timeTk {} ;
  size_t merger_size {};
  // compute the upper bound on the beta integral
  const double integral_upper_bound_M =
  (param_K > 0.0 ?
  param_K/(param_K + 1 + pow(2.0, 1.0 - param_alpha)/(param_alpha - 1.0))
                                                                  : 1.0);
  size_t new_block {} ;
  std::vector<double> v_branch_lengths (n + 1, 0.0);
  std::vector<double> v_ebib(n, 0.0);
  std::vector< size_t> tree__array__ (n, 1);
  double * prob_heads = (double *)malloc( sizeof(double));
  while( --trials > 0){
    /* run lots of experiments */
    tree__array__.clear();
    tree__array__.assign( n, 1);
   // initialise branch lengths vector
    std::fill( std::begin( v_branch_lengths),
           std::end( v_branch_lengths), 0.0);
   // generate a tree
    while( tree__array__.size() > 1 ){
      // sample time Tk
      timeTk =  sample_interval_time(
                static_cast<double>( tree__array__.size() ),
                param_alpha, integral_upper_bound_M,  prob_heads );
      // update branch lengths
      updateBranchLengths(v_branch_lengths,
                    tree__array__,  timeTk);
      // sample merger size
      merger_size = samplek( tree__array__.size(),
                                  prob_heads[0] );
      // merge blocks and update tree
      std::shuffle( std::begin( tree__array__),
```

```
                std::end(tree__array__), rng) ;
      new_block = sum_merging_blocks( tree__array__,
                                      merger_size);
      // remove the merged blocks from the tree
      tree__array__.erase( std::begin(tree__array__)
             +  tree__array__.size() -  merger_size,
                        std::end( tree__array__) );
      tree__array__.push_back( new_block);
    }
    // generated one tree
    assert( tree__array__.size() == 1);
    // add to estimate of E[B_i(n)/B(n)]
    update_estimate_ebib(v_ebib, v_branch_lengths, n);
  }

  // write out estimate of E[B_i(n)/B(n)]
  for( const auto &y: v_ebib){
    std::cout << y/static_cast<double>(number_experiments)
                                                  << '\n'; }

  // free memory
  free( prob_heads) ;
}
```

13 The main function

The `main` function for calling the simulator in § 12.

```
int main(int argc, char * argv[])
{
// initialise the GSL random number generator
  setup_rng( static_cast<size_t>(atoi(argv[4])) );

  estimate_ebib(atoi(argv[1]),
                atof(argv[2]),
                atof(argv[3]));

// free the GSL random number generator
  gsl_rng_free( rngtype);
  return 0;
}
```

References

[1] JA Chetwyn-Diggle, Bjarki Eldon, and Alison M. Etheridge. Beta-coalescents when sample size is large. in preparation.

[2] Jere Koskela. Multi-locus data distinguishes between population growth and multiple merger coalescents. *Stat. Appl. Genet. Mol. Biol.*, 17(3):20170011, 21, 2018.

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