Exact expected branch lengths for Beta-coalescents

Bjarki Eldon[1] [2]

September 3, 2021

Abstract

This code computes exact expected branch lengths for Beta-coalescents using recursions [(1)]. The incomplete Beta-coalescent is derived in [(2)]; if you use the results and/or the code please cite [(2)] when it comes out.

1 Copyright

[1]MfN Berlin, Germany
September 3, 2021
[2]Supported by Deutsche Forschungsgemeinschaft (DFG) - Projektnummer 273887127 through DFG SPP 1819: Rapid Evolutionary Adaptation grant STE 325/17-2 to Wolfgang Stephan; acknowledge funding by 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.

2 Compilation, output and execution

This CWEB (?) document (the `.w` file) can be compiled with `cweave` to generate a `.tex` file, and with `ctangle` to generate a `.c` (?) 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 (GSL).

Use `splint` to check the code:

```
splint ebikplusbeta_cweb.c
```

Use `valgrind` to check for memory leaks:

```
valgrind -v -leak-check=full -show-leak-kinds=all <program call>
```

3 introduction

We consider Λ-coalescents with transition rates, for $1 < \alpha < 2$ and $2 \le k \le n$, and $C > 0$ is a constant of proportionality,

$$\lambda_{n,k} = \binom{n}{k} \left(\mathbb{1}_{\{k=2\}} + \frac{C}{B(x; 2-\alpha, \alpha)} \int_0^x t^{k-\alpha-1}(1-t)^{n-k+\alpha-1} dt \right), \tag{1}$$

where $B(x; 2-\alpha, \alpha) = \int_0^x t^{1-\alpha}(1-t)^{\alpha-1}dt$, $0 < x \le 1$. We restrict to the case $1 < \alpha < 2$ and [2]

$$x = \frac{K}{K + m_\infty} \tag{2}$$

where $K > 0$ is a constant and

$$m_\infty = 1 + \frac{2^{1-\alpha}}{\alpha - 1}. \tag{3}$$

Let $B_i(n)$ denote the random length of branches supporting $i \in \{1, \ldots, n-1\}$ leaves. The code computes the exact expected values $\mathbb{E}\left[B_i(n)\right]$ using recursions [1].

4 Code

4.1 Includes

The included 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_pow_int.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_sf_elementary.h>
#include <gsl/gsl_sf_gamma.h>
#include <gsl/gsl_fit.h>
#include <gsl/gsl_multifit_nlin.h>
#include <gsl/gsl_sf_exp.h>
#include <gsl/gsl_sf_log.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 13.

4.2 The beta function

Compute the (incomplete) beta function

$$B(x;a,b) := \int_0^x t^{a-1}(1-t)^{b-1}dt \tag{4}$$

for $0 < x \le 1$ and $a, b > 0$.

6 ⟨beta function 6⟩ ≡

```
/* return the (incomplete) beta function */
static double betafunc(const double x, const double a, const double b)
{   /* the GSL incomplete beta function is normalised by the complete beta function */
    return (x < 1. ? gsl_sf_beta_inc(a, b, x) * gsl_sf_beta(a, b) : gsl_sf_beta(a, b));
}
```

This code is used in chunk 13.

4.3 The Beta-coalescent rate

Compute the total Beta-coal rate of merging blocks as in Eq (1).

7 ⟨betarate 7⟩ ≡

```
static double lbetank(const size_t n, const size_t j, const double a, const double K)
{      /* compute beta rate ; a is α
         */      /* n is current number of blocks
         */      /* j is number of blocks to merge
         */
    assert(n > 1);
    assert(j > 1);
    assert(j ≤ n);      /* compute the cutoff Eq (2)
         */
    double x = (K > 0. ? K/(K + (1 + pow(2, 1. − a)/(a − 1.))) : 1.);
        /* compute the Beta-part of Eq (1)
         */
    return (gsl_sf_choose((unsigned) n, (unsigned) j) * betafunc(x, ((double)
         j) − a, ((double) n) + a − ((double) j))/betafunc(x, 2. − a, a));
}
```

This code is used in chunk 13.

4.4 The jump rate

Return the jump rate of jumping from i to j blocks using the rate in Eq (1).

8 ⟨jump rate 8⟩ ≡

```
static double qij(const size_t i, const size_t j, const double a, const double K, const
      double Cconst)
{   /* compute jump rate of block counting process */   /* a is α
    */   /* K is the cutoff constant K
    */   /* Cconst is the constant of proportionality C
    */
  assert(i > 1);
  assert(j < i);
  assert(j > 0);   /* take Cconst = 1 in case of no Kingman part
    */
  return ((Cconst * lbetank(i, i - j + 1, a, K)) + (1. * (i - j ≡ 1 ? gsl_sf_choose((unsigned)
    i, 2) : 0.)));
}
```

This code is used in chunk 13.

4.5 The jump matrices

Compute the matrices of jump rates and probabilities needed for the recursions[(1)].

9 ⟨ compute jump matrices 9 ⟩ ≡

```
static void QP(const size_t n, const double a, const double K, const double
        Cconst, gsl_matrix * Q, gsl_matrix * P)
{     /* compute matrices qij and pij
       */
  size_t 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) {     /* compute the jump rate qij 4.4
        */
      x = qij(i, j, a, K, Cconst);
      s = s + x;
      gsl_matrix_set(Q, i, j, x);
      gsl_matrix_set(P, i, j, x);
    }
    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 13.

4.6 Compute the g matrix

10 ⟨compute g matrix 10⟩ ≡

```c
static void gmatrix(const size_t n, gsl_matrix *G, gsl_matrix *Q, gsl_matrix *P)
{
  size_t i, k, m;
  double s = 0.0;     /* initialise the diagonal */
  for (i = 2; i <= n; ++i) {
    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 = 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 13.

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

11 $\langle$ compute pnkb 11 $\rangle \equiv$ /* compute the matrix $p^{(n)}[k,b]$ */

```
static void pnb(const size_t n, gsl_matrix *lkb, gsl_matrix *G, gsl_matrix *P)
{   /* j is nprime from Prop A1 in paper */   /* lnb is the matrix p^(nprime)[k,b]; used for each fixed k */
  gsl_matrix *lnb = gsl_matrix_calloc(n + 1, n + 1);
  size_t 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++) {
```

$gsl\_matrix\_set(lkb, k, j, gsl\_matrix\_get(lnb, n, j));$

}

$gsl\_matrix\_set\_zero(lnb);$

} /* free the *lnb* matrix */

$gsl\_matrix\_free(lnb);$

}

This code is used in chunk 13.

4.8 compute the expected spectrum

Compute the exact expected spectrum

12 ⟨ compute ebi 12 ⟩ ≡

```
static void compute_ebi(const size_t n, const double a, const double K, const double
        Cconst)
{     /* n is sample size; number of leaves
   */     /* a is α
   */     /* K is the cutoff constant K
   */     /* Cconst is the constant of proportionality C
   */
  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);
     /* compute the Q and P matrices § 4.5
   */
  QP(n, a, K, Cconst, Q, P);     /* compute the g matrix § 4.6
   */
  gmatrix(n, G, Q, P);     /* compute pnb matrix § 4.7
   */
  pnb(n, Pn, G, P);
  size_t b, k;
  double s = 0.0;
  double eb = 0.0;
  double *v_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));
    }
    v_ebi[b] = s;
    eb = eb + s;
  }    /* print out E[B_i(n)]/E[B(n)]
        */
  for (b = 1; b < n; ++b) {
    printf("%g\n", v_ebi[b]/eb);
  }    /* free memory
        */
  gsl_matrix_free(P);
  gsl_matrix_free(Q);
  gsl_matrix_free(G);
  gsl_matrix_free(Pn);
  free(v_ebi);
}
```

This code is used in chunk 13.

4.9 the main module

The *main* function for calling *compute_ebi* § 4.8 .

13

```
⟨ Includes 5 ⟩     /* see §4.1
     */
⟨ beta function 6 ⟩     /* §4.2
     */
⟨ betarate 7 ⟩     /* §4.3
     */
⟨ jump rate 8 ⟩     /* §4.4
     */
⟨ compute jump matrices 9 ⟩     /* §4.5
     */
⟨ compute g matrix 10 ⟩     /* §4.6
     */
⟨ compute pnkb 11 ⟩     /* §4.7
     */
⟨ compute ebi 12 ⟩     /* §4.8
         */
     int main(int argc, char *argv[ ])
     {     /* § 4.8
          */
       compute_ebi((size_t) atoi(argv[1]), atof(argv[2]), atof(argv[3]), atof(argv[4]));
       return GSL_SUCCESS;
     }
```

5 references

References

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

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

Index

List of Refinements