Viability selection and high fecundity time to reach high frequency

CWEB technical report
bjarki eldon
Museum für Naturkunde
Leibniz Institut für Evolutions- und Biodiversitätsforschung
Berlin, Germany

April 12, 2017

Abstract

This code simulates viability selection in a haploid population characterised by high fecundity and sweepstakes reproduction (HFSR). An estimate of the expected time of an allelic type to reach a given frequency, conditional on the event of reaching high frequency. We exclude mutation. This CWEB (Knuth and Levy, 1994) technical report describes corresponding C (Kernighan and Ritchie, 1988) code. CWEB documents may be compiled with `cweave` and `ctangle`.

Contents

1 Copyright

2 Introduction

Some populations are highly fecund broadcast spawners and may be characterised by Type III survivorship curve. The reproduction mode of such populations has been described as sweepstakes reproduction where few parents contribute most of the offspring to a new generation. Reproduction models which take into account sweepstakes reproduction do so through heavy-tailed, or skewed, offspring distributions. The impact of such reproduction modes on selection has been little discussed (Der *et al.*, 2012; Foucart, 2013; Etheridge *et al.*, 2010).

We consider a new model of HFSR in a haploid population of fixed size N. In each generation, individual i for $i \in [N] := \{1, 2, \ldots, N\}$ for $N \in \mathbb{N} := \{1, 2, \ldots\}$ independently contributes a random number X_i of juveniles. If the total count of juveniles exceeds N random sampling of juveniles takes place in which N juveniles are sampled to form the new set of adults. In case of a highly fecund population with sweepstakes reproduction (HFSR population), the distribution of X_i is heavy-tailed with parameters $\alpha, C, \gamma > 0$ and mass function

$$\mathbb{P}(X = k) := C\left(\frac{1}{k^\alpha} - \frac{1}{(k+1)^\alpha}\right), \quad 1 \le k \le \gamma. \tag{1}$$

One can choose C so that $\mathbb{P}(X_1 = 0) \ge 0$ and $\mathbb{E}[X_1] > 1$. Our main requirement is that $\mathbb{E}[X_1] > 1$ since then the total number of juveniles is at least N with high probability for large N.

We model viability selection as follows. We assume there are n allelic types segregating in the population; we label these types by the typespace $E = \{0, 1, \ldots, n-1\}$. The juveniles inherit the types of their parents since we exclude mutation. We assume there is a *trait function* which maps the genetic type to a trait value. We assume the trait function

$$z(i) = \frac{i}{i+1}, \quad i \in \{0, 1, \ldots, n-1\}. \tag{2}$$

We assume there is a *fitness function* which maps the trait value to a fitness value. We consider an exponential fitness function, where s denotes the strength of selection and z_0 the optimal trait value,

$$w(z) = \exp\left(-s(z - z_0)^2\right), \quad z \in [0, 1]; \tag{3}$$

and an algebraic fitness function

$$w(z) = \frac{1}{1 + s(z - z_0)}, \quad z \in [0, 1]. \tag{4}$$

If the count of juveniles is greater than N we draw a random exponential with rate $w(z)$ from either the algebraic (4) or exponential (3) fitness function. The N juveniles with smallest times then form the new set of adults. If the count of juveniles equals N then all juveniles survive; we draw a new set of juveniles in case the count is less than N.

Let Y_r denote the frequency of the type conferring highest fitness at time (generation) r. Define $T := \inf\{r \in \mathbb{N} : Y_r \geq y\}$ as the first time Y_r is at least y; ie. the fittest type has reached frequency y. We estimate the conditional expected time $\tau := \mathbb{E}\left[T : Y_r > 0 \;\forall\; r\right]$. For comparison with our HFSR model (1) we model the number of juveniles according to a Poisson distribution with mean $\mathbb{E}^{(\mathrm{HFSR})}\left[X_1\right]$.

3 Compile and run

Use `cweave` on the `.w` file to generate `.tex` file, and `ctangle` to generate a `.c` file.

The necessary parameters have preset values (see section 4.7). By way of example, assuming the executable is `a.out`, then with random seed `12345` the command

```
./a.out 12345 out.out
```

writes into `out.out`

```
24
10
22
15
15
31
6
9
14
10
```

ten realisations of the time T.

4 Code

4.1 Random number generator

A random number generator of choice is declaired using the GSL_RNG_TYPE environment variable. The default generator is the 'Mersenne Twister' random number generator **?** as implemented in GSL.

5 ⟨random number generator 5⟩ ≡

declare the random number generator *rngtype*

gsl_rng ∗ *rngtype*;

Define the function *setup_rng* which initializes *rngtype*:

void *setup_rng*(**unsigned long int** *seed*)

{

set the type as *mt*19937

rngtype = *gsl_rng_alloc*(*gsl_rng_mt19937*);

gsl_rng_set(*rngtype*, *seed*);

gsl_rng_env_setup( );

}

This code is used in chunk 11.

4.2 Definitions

6 ⟨object definitions 6⟩ ≡

```
#define MAX_JUVENILES 10000000
```

This code is used in chunk 11.

4.3 Draw values for X_i

Draw values for X_i; the diploid juveniles. We take $C, \alpha > 0$ and consider

$$\mathbb{P}(X_i = k) = Ck^{-\alpha} - C\mathbb{1}(k < \psi)(1+k)^{-\alpha}, \quad k \in [\psi],$$

and we observe that $\mathbb{P}(X_i = 0) = 1 - C$. To have the mean $\mathbb{E}[X_1] > 1$ we require approximately $C > \alpha - 1$.

7 $\langle$ initialize distribution for X_i 7 $\rangle \equiv$

```
void drawXi(int N, int psi, double a, double b, gsl_ran_discrete_t *Pmass, int
        *tXi, gsl_rng *r)
{
  int k, teljari;
  tXi[0] = 0;
  teljari = 0;
  while ((tXi[0] < N) ∧ (teljari < 1000000)) {
    teljari = teljari + 1;
    tXi[0] = 0;
    for (k = 1; k ≤ N; k++) {
      tXi[k] = (a > 0. ? (int) gsl_ran_discrete(r, Pmass) : gsl_ran_poisson(r, b));
      tXi[0] = tXi[0] + tXi[k];
    }
  }
  assert(teljari < 1000000);
  assert(tXi[0] ≤ MAX_JUVENILES);
}
```

This code is used in chunk 11.

4.4 Update population

Update population given numbers x_i of juveniles generated by each individual.

8 $\langle$ population update 8 $\rangle \equiv$

double *update_population*(**int** N, **double** *variance*, **int** *nalleles*, **double** s, **double** *znull*, **double** *epsilon*, **int** $*$*Pop*, **int** $*$*tXi*, **int** $*$*tempJuve*, **double** $*Z$, **double** $*$*locuseffects*, **double** $*$*etimes*, **size_t** $*$*aindex*, *gsl_rng* $*\, r$)

{

/$*$ N is number of pairs, L is number of loci, s is selection coefficient, *znull* is trait optimum $*$/

int i, k, *xindex*;

double *Zbar* $= 0.$;

double w;

/$*$ $tXi[0] = X_1 + \cdots + X_N$ is the total number of juveniles $*$/

xindex $= 0$;

for $(i = 1;\ i \le N;\ i{+}{+})$ {

/$*$ check if individual i produced potential offspring, ie. if $X_i > 0$ $*$/

if $(tXi[i] > 0)$ {

for $(k = 1;\ k \le tXi[i];\ k{+}{+})$ {

/$*$ if no mutation, copy the type of the parent $*$/

tempJuve[*xindex* + 1] = *Pop*[i];

/$*$ compute the trait value $z_i = \mathbb{1}\,(v > 0)\, G(0, v) + \frac{i}{1+i}$ of juvenile i where v denotes the variance $*$/

Z[*xindex* + 1] = (*variance* > 0. ? *gsl_ran_gaussian_ziggurat*(r, *variance*) : 0.) + *locuseffects*[*tempJuve*[*xindex* + 1]];

/$*$ compute fitness value $w = 1/(1 + s(z_i - z_0)^2)$; now exponential fitness function $w = \exp\left(-s(z_i - z_0)^2\right)$ $*$/

w = 1./(1. + (s $*$ *gsl_pow_2*(Z[*xindex* + 1] − *znull*)));

assert($w > 0$); /$*$ draw exponential times with rate w $*$/

```
          etimes[xindex] = gsl_ran_exponential(r, 1./w);
          xindex = xindex + 1;
        }
      }
    }
    assert(xindex ≡ tXi[0]);
      /* sort the exponential times, if tXi[0] > N */
    if (tXi[0] > N) {
      gsl_sort_index(aindex, etimes, 1, tXi[0]);
        /* the first N indexes in aindex are the indexes of the surviving juveniles */
      for (i = 0; i < N; i++) {
        Pop[i + 1] = tempJuve[aindex[i]];
          /* the trait value of the population is given by z̄ = (1/N) Σ_i z_σ(i) where σ(i) is the
             ordered index i, in ascending order of the associated exponential times; we
             compute and return the fraction of the null type, the most fit type */
          /* Zbar counts the number of alleles of the fittest type; either type n − 1 or 0
             can be the fittest types */
        Zbar = Zbar + (znull > 0. ? (Pop[i + 1] < nalleles − 1 ? 0.0 : 1.0) : (Pop[i + 1] > 0 ?
             0. : 1.0))/((double) N);
      }
    }
    else {
        /* exactly N juveniles, so all survive */
      for (i = 0; i < N; i++) {
        Pop[i + 1] = tempJuve[i + 1];
          /* Zbar counts the number of alleles of the fittest type; either type n − 1 or 0
             can be the fittest types */
        Zbar = Zbar + (znull > 0. ? (Pop[i + 1] < nalleles − 1 ? 0.0 : 1.0) : (Pop[i + 1] > 0 ?
             0. : 1.0))/((double) N);
      }
    }
    return (Zbar);
  }
```

This code is used in chunk 11.

4.5 Simulator

Run many replicates.

9 ⟨ replicates 9 ⟩ ≡

```
void simulator(int N, int nalleles, double variance, double a, double b, int Psi, double
        s, double znull, double epsilon, int nruns, char skra[200], gsl_rng *r)
{
    /* N is population size; nalleles is number of alleles */
  double zbar;
  int *Pop = (int *) calloc(N + 1, sizeof(int));
  double *Z = (double *) calloc(MAX_JUVENILES, sizeof(double));
  size_t *aindex = (size_t *) calloc(MAX_JUVENILES, sizeof(size_t));
  double *etimes = (double *) calloc(MAX_JUVENILES, sizeof(double));
  int *tempJuve = (int *) calloc(MAX_JUVENILES, sizeof(int));
  double *PXi = (double *) calloc(1 + Psi, sizeof(double));
  double *leffects = (double *) calloc(nalleles, sizeof(double));
  double X0;
  int *tXi = (int *) calloc(N + 1, sizeof(int));
  int k, ngens;
  double mean = 0.;
  PXi[0] = 0.;
  for (k = 1; k <= Psi; k++) {
      /* P(X_i = k) = k^{-α} − (k + 1)^{-α} for 1 ≤ k ≤ ψ */
    PXi[k] = (a > 0. ? (pow(1./(((double) k)), a) - pow(1./(((double)(1 + k))), a)) : 1.);
    assert(PXi[k] >= 0.);
    mean = mean + (((double) k) * PXi[k]);
  }
  gsl_ran_discrete_t *Pmass = gsl_ran_discrete_preproc(1 + Psi, PXi);
      /* here we set the allelic type effects ξ_j; one option might be ξ_j = j/(1 + j), another
      option might be ξ_j = j/n where n is the number of types, and 0 ≤ j ≤ n − 1. */
  for (k = 0; k < nalleles; k++) {
    leffects[k] = ((double) k)/((double)(1 + k));
  }
  int rep = 0;
  while (rep < nruns) {
```

```
zbar = 0.;
    /* initialise population by assigning allelic type from {0,1,...,n-1}, where
    n is number of types, uniformly at random to each individual. Initialise
    zbar = z̄ = (1/N) Σ_i z_i where z_i = 1(σ > 0) N(0,σ) + ξ_i(g_i) where g_i is the genotype
    of individual i, and N(0,σ) is a random Gaussian with mean 0 and variance σ */
X0 = 0.0;
for (k = 1; k ≤ N; k++) {
    /* assign a type modulo n where n is number of types; set ℝ_n := {0,1,...,n-1}
      and we assign type a_j = j mod n where a_j ∈ ℝ_n */
  Pop[k] = (int)(k % nalleles);
      /* starts almost fixed at type n-1; one copy of each of other alleles */
  assert(Pop[k] ≥ 0);
  assert(Pop[k] < nalleles);
  X0 = X0 + (znull > 0. ? (Pop[k] < nalleles - 1 ? 0.0 : 1.0) : (Pop[k] > 0 ? 0. :
      1.))/((double) N);
  zbar = zbar + (variance > 0. ? gsl_ran_gaussian_ziggurat(r,
      variance) : 0.) + leffects[Pop[k]];
}
ngens = 0;
    /* ε is the fraction of the null type - the most fit type */
while (((ngens < 100000) ∧ (X0 < epsilon)) ∧ (X0 > 0.0)) {
  drawXi(N, Psi, a, b, Pmass, tXi, r);
  zbar = update_population(N, variance, nalleles, s, znull, epsilon, Pop, tXi,
      tempJuve, Z, leffects, etimes, aindex, r);
  ngens = ngens + 1;
  X0 = zbar;
}
if (X0 > 0.0) {
  FILE *f = fopen(skra, "a");
  fprintf(f, "%d\n", (X0 > 0 ? ngens : -1));
  fclose(f);
}
rep = rep + (X0 > 0.0 ? 1 : 0);
}
    /* free memory */
free(Z);
```

```
    free(tXi);
    free(PXi);
    gsl_ran_discrete_free(Pmass);
    free(tempJuve);
    free(etimes);
    free(aindex);
    free(Pop);
    free(leffects);
}
```

This code is used in chunk 11.

4.6 run over parameters

Run over some parameters.

10 ⟨ parameters 10 ⟩ ≡

```
void run_parameters(int N, int nalleles, double variance, double a, double
        b, int Psi, double s, double znull, double epsilon, int nruns, char
        skra[200], gsl_rng *r)
{
  double ai = 1.;
  double out;
  int psii;
  while (ai < 2.) {
    for (psii = 100000; psii < 1000001; psii = psii + 100000) {
      out = new_simulator(N, nalleles, variance, (ai > 0. ? ai : 0.), b, psii, s, znull,
          epsilon, nruns, r);
      printf("%g %d %g\n", ai, psii, out);
      FILE *f = fopen(skra, "a");
      fprintf(f, "%g ", ai);
      fprintf(f, "%d ", psii);
      fprintf(f, "%g\n", out);
      fclose(f);
    }
    ai = ai + 0.1;
  }
}
```

4.7 the *main* function

11 ⟨ Includes 12 ⟩

⟨ random number generator 5 ⟩

⟨ object definitions 6 ⟩

⟨ initialize distribution for X_i 7 ⟩

⟨ population update 8 ⟩

⟨ replicates 9 ⟩

```
int main(int argc, char *argv[])
{
 initialise the random number generator
  setup_rng((unsigned long int) atoi(argv[1]));
```

/* the expected value $\mathbb{E}[X] = 11.09016$ when $(\alpha, \gamma) = (1.0, 10^5)$; $\mathbb{E}[X] = 2.606051$ when $(\alpha, \gamma) = (1.5, 10^5)$; $\mathbb{E}[X] = 1.644924$ when $(\alpha, \gamma) = (2.0, 10^5)$; $\mathbb{E}[X] = 6.48$ when $(\alpha, \gamma) = (1.0, 10^3)$; */
/* `POP_SiZE` is N; `N_ALLELES` is number of alleles; `PSI_TRUNCATION` is γ in (1); `TRAIT_OPTIMUM` is z_0 (4), (3); `EPSILON` is y the threshold frequency */

```
#define POP_SIZE 1000
#define N_ALLELES 100
#define VARIANCE 0.0
```

/* If `ALPHA` (α) is 0 then the Poisson distribution is assumed with mean `BETA` */

```
#define ALPHA 1.0
#define BETA 7.485471
#define PSI_TRUNCATION 100000
#define S_SELECTION 1.0
#define TRAIT_OPTIMUM 0.0
#define EPSILON 0.95
#define RUNS 10
  simulator(POP_SIZE, N_ALLELES, VARIANCE, ALPHA, BETA, PSI_TRUNCATION,
     S_SELECTION, TRAIT_OPTIMUM, EPSILON, RUNS, argv[2], rngtype);
  gsl_rng_free(rngtype);
  return GSL_SUCCESS;
}
```

5 Includes

12 ⟨Includes 12⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <math.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_integration.h>
#include <gsl/gsl_sf_exp.h>
#include <gsl/gsl_sf_log.h>
#include <gsl/gsl_sf_expint.h>
#include <gsl/gsl_combination.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_combination.h>
#include <gsl/gsl_statistics_double.h>
#include <gsl/gsl_statistics_int.h>
#include <gsl/gsl_sort.h>
#include <assert.h>
```

This code is used in chunk 11.

6 References

References

Der, R., C. Epstein, and J. B. Plotkin, 2012 Dynamics of neutral and selected alleles when the offspring distribution is skewed. Genetics **191**: 1331–1344.

Etheridge, A. M., R. C. Griffiths, and J. E. Taylor, 2010 A coalescent dual process in a Moran model with genic selection, and the Lambda coalescent limit. Theor Popul Biol **78**: 77–92.

Foucart, C., 2013 The impact of selection in the λ-wright-fisher model. Electron. Commun. Probab **18**: 1–10.

Kernighan, B. W., and D. M. Ritchie, 1988 The c programming language.

Knuth, D. E., and S. Levy, 1994 *The CWEB system of structured documentation: version 3.0*. Addison-Wesley Longman Publishing Co., Inc.

7 Funding

Funded by DFG grant 325/17-1 to Wolfgang Stephan through DFG SPP Priority Programme 1819: Rapid Evolutionary Adaptation (`https://dfg-spp1819.uni-hohenheim.de/en/105254`).

Index

List of Refinements

⟨ Includes 12 ⟩ Used in chunk 11.
⟨ initialize distribution for X_i 7 ⟩ Used in chunk 11.
⟨ object definitions 6 ⟩ Used in chunk 11.
⟨ parameters 10 ⟩
⟨ population update 8 ⟩ Used in chunk 11.
⟨ random number generator 5 ⟩ Used in chunk 11.
⟨ replicates 9 ⟩ Used in chunk 11.