Viability selection and high fecundity

CWEB technical report

bjarki eldon

Museum für Naturkunde

Leibniz Institut für Evolutions- und Biodiversitätsforschung

Berlin, Germany

April 12, 2017

Abstract

This CWEB (?) technical report describes corresponding C (?) code. CWEB documents may be compiled with `cweave` and `ctangle`.

Contents

1 Introduction

dffdf

2 Compile and run

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

3 Code

3.1 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.

4 ⟨random number generator 4⟩ ≡

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 $mt19937$

```
    rngtype = gsl_rng_alloc(gsl_rng_mt19937);
    gsl_rng_set(rngtype, seed);
    gsl_rng_env_setup();
}
```

This code is used in chunk 10.

3.2 Definitions

5 ⟨object definitions 5⟩ ≡

```
#define MAX_JUVENILES 10000000
```

This code is used in chunk 10.

3.3 Draw values for X_i

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

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

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

6 ⟨initialize distribution for X_i 6⟩ ≡

```
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 10.

3.4 Update population

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

7 $\langle$ population update 7 $\rangle \equiv$

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 */

$etimes[xindex] = gsl\_ran\_exponential(r, 1./w);$

$xindex = xindex + 1;$

}

}

}

$assert(xindex \equiv tXi[0]);$

/* 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 $\overline{z} = \frac{1}{N}\sum_i z_{\sigma(i)}$ where $\sigma(i)$ 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 */

$Zbar = Zbar + (znull > 0.\ ?\ (Pop[i+1] < nalleles - 1\ ?\ 0.0 : 1.0) : (Pop[i+1] > 0\ ?\ 0. : 1.0))/((\textbf{double})\ N);$

```
    }
  }
  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 10.

3.5 Simulator

Run many replicates.

8 ⟨ replicates 8 ⟩ ≡

```
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 $\xi_j = j/(1+j)$, another option might be $\xi_j = j/n$ where n is the number of types, and $0 \leq j \leq n-1$. */

for $(k = 0;\ k < nalleles;\ k{+}{+})$ {

$leffects[k] = ((\textbf{double})\ k)/((\textbf{double})(1 + k));$

}

int $rep = 0;$

while $(rep < nruns)$ {

$zbar = 0.;$

/* initialise population by assigning allelic type from $\{0, 1, \ldots, n-1\}$, where n is number of types, uniformly at random to each individual. Initialise $zbar = \overline{z} = \frac{1}{N}\sum_i z_i$ where $z_i = \mathbb{1}(\sigma > 0)\, N(0, \sigma) + \xi_i(g_i)$ where g_i is the genotype of individual i, and $N(0, \sigma)$ is a random Gaussian with mean 0 and variance σ */

$\texttt{X0} = 0.0;$

for $(k = 1;\ k \leq N;\ k{+}{+})$ {

/* assign a type modulo n where n is number of types; set $\mathbb{R}_n := \{0, 1, \ldots, n-1\}$ and we assign type $a_j = j \mod n$ where $a_j \in \mathbb{R}_n$ */

$Pop[k] = (\textbf{int})(k \,\%\, nalleles);$

/* starts almost fixed at type $n-1$; one copy of each of other alleles */

$assert(Pop[k] \geq 0);$

$assert(Pop[k] < nalleles);$

$\texttt{X0} = \texttt{X0} + (znull > 0.\ ?\ (Pop[k] < nalleles - 1\ ?\ 0.0 : 1.0) : (Pop[k] > 0\ ?\ 0. : 1.))/((\textbf{double})\ N);$

$zbar = zbar + (variance > 0.\ ?\ gsl\_ran\_gaussian\_ziggurat(r, variance) : 0.) + leffects[Pop[k]];$

```
    }
    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 10.

3.6 run over parameters

Run over some parameters.

9 ⟨ parameters 9 ⟩ ≡

```
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;
  }
}
```

3.7 the *main* function

10 ⟨Includes 11⟩

⟨random number generator 4⟩

⟨object definitions 5⟩

⟨initialize distribution for X_i 6⟩

⟨population update 7⟩

⟨replicates 8⟩

```
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 $(\alpha,\gamma) = (1.0, 10^3)$; */

```
#define POP_SIZE 100000
#define N_ALLELES 100
#define VARIANCE 0.0
#define ALPHA 0.0
#define BETA 7.485471
#define PSI_TRUNCATION 1000
#define S_SELECTION 0.1
#define TRAIT_OPTIMUM 1.0
#define EPSILON 0.95
#define RUNS 200
  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;
}
```

4 Includes

11 ⟨Includes 11⟩ ≡

```
#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 10.

Index

List of Refinements