Discrete-time simulator for multi-loci selection on a trait

CWEB technical report

bjarki eldon

Museum für Naturkunde

Leibniz Institut für Evolutions- und Biodiversitätsforschung

Berlin, Germany

August 25, 2016

Abstract

Simulates the time (number of generations) it takes a diploid population to reach a predefined optimum trait value. Selection acts on the (multi-locus) trait, and we compare the time to reach optimum for two reproduction schemes. This CWEB [**? ? ?**] technical report describes corresponding C code. CWEB documents may be compiled with `cweave` and `ctangle`.

Contents

1 Copyright **2**

2 Introduction **4**

3 Compile and run **6**

4 Code **7**

4.1 **Random number generator** 8

4.2 **Definitions** 9

4.3 **Draw values for X_i** 10

4.4 **Update population** 12

4.5 **Simulator** 16

4.6 **many runs** 19

4.7 **Allocate a 2d array** 21

4.8 **Free a 2d array** 22

4.9 **the *main* function** 23

4.10 Print a GSL integer matrix 25

5 Includes **26**

6 Funding **27**

7 References **28**

1 Copyright

2 Introduction

We are interested in investigating if high fecundity and skewed distribution of offspring (HFSDO), in addition to selection acting on a multi-loci trait, drives rapid adaptation. In particular, does the combination of HFSDO and selection acting on a multi-loci trait enable more rapid adaptation than in the absence of HFSDO. Also, in the single-locus case, in both haploid and diploid case, we would like to know if HFSDO enables more rapid adaptation than a non-skewed reproduction mechanism.

Our population size is constant at N. In each generation, each pair independently contributes offspring. We let X_i denote the random number of offspring of pair i, and the X_i are iid, and independent between generations. Let b, α, ψ be positive constants; define $\mathbb{1}(A) := 1$ if A holds, and zero otherwise; write $[n]_0 := \{0, 1, \ldots, n\}$ for $n \in \mathbb{N} := \{1, 2, \ldots\}$; $[n] := \{1, 2, \ldots, n\}$. The law of X_i is said to be "skewed" if it is given by

$$\mathbb{P}(X_i = k) = \frac{b^\alpha}{(b+k)^\alpha} - \frac{b^\alpha \mathbb{1}(0 \le k < \psi)}{(1+b+k)^\alpha}, \quad k \in [\psi]_0; \tag{1}$$

the parameter b shifts the distribution to allow for mass at zero. The distribution (1) is monotonically decreasing with increasing k. The law (1) is similar in form to the model by [**?**]. However, the model by [**?**] is a limit statement, and as such can only be used in theoretical calculations.

We compare the law (1) of X_i with a Poisson distribution with parameter $\lambda > 0$. For $\alpha < 2$ the law (1) has a much heavier tail than the Poisson. The population size is constant at $2N$ generations; N pairs of individuals independently contribute juveniles. The total number of juveniles generated in any given generation therefore needs to be at least $2N$. If the law of X_i is Poisson, we take $\lambda \ge 2$; otherwise the sampling takes a long time since the total pool of juveniles is regenerated until the total count of juveniles is at least $2N$.

Given the allelic types $g_{i,j}^{(1)}$ and $g_{i,j}^{(2)}$ at locus $j \in [L]$ of individual $i \in [2N]$, the trait

value of individual i is computed as

$$z_i = \frac{1}{L} \sum_{j \in [L]} \xi_j \left(g_{i,j}^{(1)} + g_{i,j}^{(2)} \right). \tag{2}$$

The trait value of the population is the average of the individual trait values.

The optimum trait value is denoted by z_0. Given the trait value z_i of individual i, the fitness is given by

$$w_i = \exp\left(-s(z_i - z_0)^2 \right). \tag{3}$$

Equation (2) for the trait value allows one, for example, to consider linear effects of allelic types. By way of example, set $\xi_j = 1$ for simplicity, and let homozygotes for type 0 be the fittest genotype, heterozygotes less fit, and homozygotes for type 1 the least fit. Then $z_0 = 0$, and if there's one locus, the possible fitness values are 1, e^{-s}, and e^{-2s}.

If, in any given generation, the number of juveniles ($\sum_i x_i$) exceeds $2N$, we draw $\sum_i x_i$ independent exponentials (T_i), where T_i has rate w_i. The $2N$ juveniles with the lowest times then form the next set of adults. The genotypes of the parents do not affect the number of juveniles contributed, rather indirectly the viability of the juveniles. The viability of a juvenile depends on its' genotypes.

3 Compile and run

Use `cweave` on the `.w` file to generate `.tex` file, and `ctangle` to generate a `.c` file. To compile an executable:

```
gcc <flags> discrete_linear.c -lm -lgsl -lgslcblas
```

The command to run simulations (assuming `a.out` is the executable) is

```
a.out <N> <L> <a> <b> <p> <s> <z> <e> <R> <r> <f>
```

where `N` is number of pairs, `L` is number of loci, `a` is parameter α; `b` is parameter b, the shift of the distribution; `p` is ψ, the offspring cutoff; `s` is the selection strength s; `z` the optimum z_0; `e` is the allowed distance ε from optimum; `R` is number of runs; `r` is a random seed, `f` is the name of the file storing the results, the time to reach optimum. If $\alpha > 0$, the law of X_i is assumed skewed; otherwise set $\alpha = 0$ to sample from the Poisson with b taken as the parameter (λ) for the Poisson.

By way of example,

```
a.out 50000 1 0. 2.1 10 10. 0. 0.000001 5 123 results.out
```

writes the following into `results.out`:

```
58
58
60
57
56
```

4 Code

The configuration of the population is stored as a $(2L) \times (2N)$ matrix, with rows denoting loci, and each column stores the types of one diploid individual. Rows 1 to L store the types an individual received from one parent, and rows $L+1$ to $2L$ store the types received from the other parent. Rows j and $j + L$ for $j \in [L]$ hold the types at locus j. The configuration of juveniles is stored in the same way.

The GSL library (see `https://www.gnu.org/software/gsl/`) is used for random number generation and sorting. If the number of juveniles exceeds $2N$ in any given generation, the exponential times of the juveniles are sorted into ascending order, and the $2N$ juveniles with the lowest times form the new set of adults. We set a maximum allowed number of juveniles for computer memory purposes.

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 $mt19937$

$rngtype = gsl\_rng\_alloc(gsl\_rng\_mt19937);$

$gsl\_rng\_set(rngtype, seed);$

$gsl\_rng\_env\_setup();$

}

This code is used in chunk 13.

4.2 Definitions

We set the maximum number of juveniles for memory purposes.

6 ⟨ object definitions 6 ⟩ ≡

```
#define MAX_JUVENILES  10000000
```

This code is used in chunk 13.

4.3 Draw values for X_i

Draw values for X_i; the diploid juveniles. We have, for $b, \alpha > 0$ some constants,

$$\mathbb{P}\left(X_i = k\right) = \frac{b^\alpha}{(b+k)^\alpha} - \frac{b^\alpha \mathbb{1}\left(0 \le k < \psi\right)}{(1+b+k)^\alpha}, \quad k \in [\psi]_0,$$

and we observe that $\sum_{0 \le k \le \psi} \mathbb{P}\left(X_i = k\right) = 1$.

7 ⟨ initialize distribution for X_i 7 ⟩ ≡

```
void drawXi(int N, int psi, double a, double b, gsl_ran_discrete_t * Pmass, int
        *tXi, gsl_rng * r)
{
    /* N is number of pairs; psi is the truncation of the skewed offspring distribution
      (1); the number of juveniles contributed by each pair is stored in tXi; Pmass is
      the distribution (1) */
  int k, teljari;
  tXi[0] = 0;
  teljari = 0;
    /* we need the total number of juveniles to be at least 2N */
  while ((tXi[0] < 2 * N) ∧ (teljari < 1000000)) {
    teljari = teljari + 1;
    tXi[0] = 0;
    for (k = 1; k ≤ N; k++) {
      /* if a > 0 we assume the skewed law (1); otherwise Poisson with parameter
        λ = b */
      tXi[k] = (a > 0. ? (int) gsl_ran_discrete(r, Pmass) : gsl_ran_poisson(r, b));
      tXi[0] = tXi[0] + tXi[k];
    }
  }
```

```
        /* assert breaks the execution if the statement does not hold */
    assert(teljari < 1000000);
    assert(tXi[0] ≤ MAX_JUVENILES);
}
```

This code is used in chunk 13.

4.4 Update population

Update population given values x_i of juveniles generated by each pair. The genotypes of the parents do not affect the number of juveniles contributed. Genotypes are assigned to each juvenile, and trait value computed. If the number of juveniles exceeds the population size $2N$, selection sorts out the $2N$ fittest (on average) individuals. We randomly pair the parents; since we are not modeling fecundity selection, we can separately draw N numbers of juveniles, and then assign to randomly formed parent pairs.

The module *juvenile_genotypes* returns the trait value of the $2N$ juveniles that form the new set of adults.

8 ⟨ assign juvenile genotypes 8 ⟩ ≡

```
double juvenile_genotypes(int N, int L, double s, double znull, double epsilon, int
        **Pop, int *indindex, int *tXi, gsl_matrix_int *tempJuve, double *Z, double
        *locuseffects, double *etimes, size_t *aindex, int *allelefr, gsl_rng *r)
{
    /* N is number of pairs; L is number of loci; s is selection coefficient; znull is
    optimum trait value; epsilon is maximum allowed distance from optimum; Pop is
    the matrix for the population configuration of types; indindex is vector of indexes
    for pairing the parents; tXi stores the N numbers of juveniles; Z stores the juvenile
    trait values; tempJuve stores the type configuration of the juveniles; locuseffects
    is the vector of the effects of loci, the ξ_j in (2); etimes stores the exponential times;
    aindex storest the sorted indexes of juveniles; allelefr stores the count of an allele
    of some type, for consistency check; r is the random number generator */

    int i, j, k, B, xindex;
    double Zbar = 0.;
    allelefr[0] = 0;
        /* tXi[0] = x_1 + ··· + x_N is the total number of juveniles */
```

$xindex = 0;$

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

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

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

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

for $(j = 1;\ j \leq 2 * L;\ j{+}{+})$ {

/* if B takes value 0 we assign type from set $[L]$, otherwise from set $\{L+1, \ldots, 2L\}$; genotypes for range $[L]$ are assigned from individual $\sigma(i)$, otherwise from individual $\sigma(N+i)$, where σ denotes the permutation of the indexes */

$B = (\textbf{int})\ gsl\_ran\_bernoulli(r, .5);$

$gsl\_matrix\_int\_set(tempJuve, j, xindex + 1, Pop[(j > L\ ?\ (B > 0\ ?\ j :$
$j - L) : (B > 0\ ?\ j + L : j))][(j > L\ ?\ indindex[N + i] : indindex[i])]);$

}

/* now have assigned genotypes for juvenile, compute trait value */

$Z[xindex] = 0.;$

for $(j = 1;\ j \leq L;\ j{+}{+})$ {

$Z[xindex] = Z[xindex] + (locuseffects[j] *$
$((\textbf{double})(gsl\_matrix\_int\_get(tempJuve, j,$
$xindex + 1) + gsl\_matrix\_int\_get(tempJuve, j + L,$
$xindex + 1)))/((\textbf{double})\ L));$

}

/* given trait value z for juvenile, compute the fitness value $w = e^{-s(z-z_0)^2}$, and draw exponential with rate w */

```
        etimes[xindex] = gsl_ran_exponential(r,
            1./gsl_sf_exp(-s * gsl_pow_2(Z[xindex] - znull)));
        xindex = xindex + 1;
      }
    }
}
assert(xindex ≡ tXi[0]);


assert(tXi[0] ≥ N + N);
if (tXi[0] > 2 * N) {
    /* sort the times into ascending order, and store the sorted indexes in aindex */


    gsl_sort_index(aindex, etimes, 1, tXi[0]);
        /* the first 2N indexes in aindex are the indexes of the surviving juveniles */
    for (i = 0; i < N + N; i++) {
      Zbar = Zbar + Z[(int) aindex[i]]/((double)(N + N));
      for (j = 1; j ≤ L + L; j++) {
        allelefr[0] = allelefr[0] + (gsl_matrix_int_get(tempJuve, j, (int)
            aindex[i]) ≡ 0 ? 1 : 0);
        Pop[j][i + 1] = gsl_matrix_int_get(tempJuve, j, (int) aindex[i]);
      }
    }
}
else {
    /* tXi[0] = 2N, so all juveniles survive */
    for (i = 0; i < N + N; i++) {
```

```
        Zbar = Zbar + Z[i]/((double)(N + N));
        for (j = 1; j ≤ L + L; j++) {
          allelefr[0] = allelefr[0] + (gsl_matrix_int_get(tempJuve, j, i) ≡ 0 ? 1 : 0);
          Pop[j][i + 1] = gsl_matrix_int_get(tempJuve, j, i + 1);
        }
      }
    }
    return (Zbar);
  }
```

This code is used in chunk 13.

4.5 Simulator

A discrete-time simulator for a single run.

9 ⟨ discrete-time simulator 9 ⟩ ≡

```
int simulator(int N, int L, double a, double b, int Psi, double s, double
        znull, double epsilon, double *leffects, gsl_ran_discrete_t * PmXi, int
        **Pop, gsl_rng * r)
{   /* P is the population; 2N is number of diploid individuals; L is number of
    unlinked loci; a is the skewness parameter; gamma = γ = (γ_1, ..., γ_L) is the vector
    of values of locus effects. First we initialize the population P. */
    /* Pop is the current genotypes of the 2N diploid individuals; tempXi is the
      random number of juveniles per parent pair; pindex is the index of individuals
      used to form pairs; */
  int *tempXi = (int *) calloc(N + 1, sizeof(int));
  size_t *jindex = (size_t *) calloc(MAX_JUVENILES, sizeof(size_t));
  int *pindex = (int *) calloc(N + N, sizeof(int));
  gsl_matrix_int * gjuveniles = gsl_matrix_int_calloc(L + L + 1, MAX_JUVENILES + 1);
  double *traits = (double *) calloc(MAX_JUVENILES, sizeof(double));
  double *Etimes = (double *) calloc(MAX_JUVENILES, sizeof(double));
  int i, j, B, iter;
  int *allfr = (int *) calloc(1, sizeof(int));
  for (i = 1; i <= 2 * N; i++) {
    pindex[i - 1] = i;
    for (j = 1; j <= 2 * L; j++) {
      /* ℙ(B = 1) = 1/2; ie. we assign initial alleles with equal probability */
      B = (int) gsl_ran_bernoulli(r, .5);
      Pop[j][i] = B;
    }
```

```
}
double zbar = 0.;
iter = 0;
for (i = 1; i <= 2 * N; i++) {
  for (j = 1; j <= L; j++) {
    zbar = zbar + (leffects[j] * (Pop[j][i] + Pop[j + L][i]));
  }
}
      /* print out the initial value of the mean trait */
zbar = zbar/((double)(N + N));
while (((iter < 100000) ∧ (fabs(zbar - znull) > epsilon))) {
    /* Pair the diploid individuals, the parents: first shuffle the index values */
  gsl_ran_shuffle(r, pindex, 2 * N, sizeof(int));
  drawXi(N, Psi, a, b, PmXi, tempXi, r);
      /* now have values of X_i for all N pairs; update population */
  zbar = juvenile_genotypes(N, L, s, znull, epsilon, Pop, pindex, tempXi, gjuveniles,
      traits, leffects, Etimes, jindex, allfr, r);
  iter = iter + 1;
      /* only need to check if all alleles are 1 */
  iter = ((allfr[0] > 0) ? iter : 1000000);
}
printf("allfr␣␣%d␣␣%d\n", allfr[0], iter);
    /* free used memory */
free(tempXi);
free(jindex);
free(pindex);
gsl_matrix_int_free(gjuveniles);
```

```
    free(allfr);
    free(traits);
    free(Etimes);
        /* return the count of generations needed to reach optimum */
    return (iter);
}
```

This code is used in chunk 13.

4.6 many runs

Generate many replicates and the time, number of generations, to optimum.

10 ⟨ many runs 10 ⟩ ≡

```
void replicates(int N, int L, double a, double b, int psi, double seleccoef, double
        znull, double epsilon, int nruns, double *loceffects, gsl_rng * r, char
        skra[200])
{
  double *PXi = (double *) calloc(1 + psi, sizeof(double));
  int **mPop = alloc_2d_array(L, N);
  int k;
  double mean = 0.;
  for (k = 0; k <= psi; k++) {
    PXi[k] = (a > 0. ? pow(b, a) * (pow(1./(((double) k) + b),
        a) - (k < psi ? pow(1./(((double)(1 + k)) + b), a) : 0.)) : 1.);
    assert(PXi[k] >= 0.);
    mean = mean + (((double) k) * PXi[k]);
  }
  assert(mean >= 2.);
  gsl_ran_discrete_t * Pmass = gsl_ran_discrete_preproc(1 + psi, PXi);
  FILE *f = fopen(skra, "w");
  int rep, svar;
  for (rep = 0; rep < nruns; rep++) {
    svar = simulator(N, L, a, b, psi, seleccoef, znull, epsilon, loceffects, Pmass, mPop, r);
    fprintf(f, "%d\n", svar);
    printf("%d\n", rep);
  }
```

```
    free(PXi);
    gsl_ran_discrete_free(Pmass);
    fclose(f);
    free_2d_array(mPop, L);
}
```

This code is used in chunk 13.

4.7 Allocate a 2d array

Allocate a 2d array; use it for large population size.

11 ⟨ allocate array 11 ⟩ ≡

```
int **alloc_2d_array(int rows, int cols)
{
  int **m = Λ;
  m = (int **) calloc(rows + rows + 1, sizeof(int *));
  int i;
  for (i = 0; i ≤ rows + rows; i++) {
    m[i] = (int *) calloc(cols + cols + 1, sizeof(int));
  }
  return (m);
}
```

This code is used in chunk 13.

4.8 Free a 2d array

Free a 2d array.

12 ⟨ free 2d array 12 ⟩ ≡

```
void free_2d_array(int **a, int n)
{
  int i;
  for (i = 0; i ≤ n + n; i++) {
    free(a[i]);
  }
  free(a);
  a = Λ;
}
```

This code is used in chunk 13.

4.9 the *main* function

13 ⟨ Includes 15 ⟩

⟨ print matrix 14 ⟩

⟨ allocate array 11 ⟩

⟨ free 2d array 12 ⟩

⟨ random number generator 5 ⟩

⟨ object definitions 6 ⟩

⟨ initialize distribution for X_i 7 ⟩

⟨ assign juvenile genotypes 8 ⟩

⟨ discrete-time simulator 9 ⟩

⟨ many runs 10 ⟩

```
int main(int argc, char *argv[])
{
 initialise the random number generator
  setup_rng((unsigned long int) atoi(argv[10]));
  double *leffects = (double *) calloc(atoi(argv[2]) + 1, sizeof(double));
  int ell;
  for (ell = 1; ell ≤ atoi(argv[2]); ell++) {
    leffects[ell] = 1.;
  }
  leffects[1] = 1.;
  replicates(atoi(argv[1]), atoi(argv[2]), atof(argv[3]), atof(argv[4]), atoi(argv[5]),
      atof(argv[6]), atof(argv[7]), atof(argv[8]), atoi(argv[9]), leffects, rngtype,
      argv[11]);
   // clear all used memory:
  gsl_rng_free(rngtype);
```

```
    free(leffects);
    return GSL_SUCCESS;
     end of main function
}
```

4.10 Print a GSL integer matrix

print matrix

14 ⟨ print matrix 14 ⟩ ≡

```
void printmatrix(int rows, int cols, gsl_matrix_int * m)
{
  int i, j;
  for (i = 1; i ≤ rows; i++) {
    for (j = 1; j ≤ cols; j++) {
      printf("%d␣", gsl_matrix_int_get(m, i, j));
    }
    printf("\n");
  }
}
```

This code is used in chunk 13.

5 Includes

15 ⟨ Includes 15 ⟩ ≡

```
#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 13.

6 Funding

Funding provided by DFG grant STE 325/17-1 to Wolfgang Stephan through the DFG Priority Programme SPP 1819 "Rapid Evolutionary Adaptation" (https://dfg-spp1819.uni-hohenheim.de/105254?L=1). The generous support of Museum für Naturkunde in Berlin is warmly acknowledged.

7 References

Index

a: 7, 9, 10, 12.

aindex: 8.

allelefr: 8.

allfr: 9.

alloc_2d_array: 10, 11.

argc: 13.

argv: 13.

assert: 7, 8, 10.

atof: 13.

atoi: 13.

B: 8, 9.

b: 7, 9, 10.

calloc: 9, 10, 11, 13.

cols: 11, 14.

drawXi: 7, 9.

ell: 13.

epsilon: 8, 9, 10.

etimes: 8.

Etimes: 9.

f: 10.

fabs: 9.

fclose: 10.

fopen: 10.

fprintf: 10.

free: 9, 10, 12, 13.

free_2d_array: 10, 12.

gjuveniles: 9.

gsl_matrix_int: 8, 9, 14.

gsl_matrix_int_calloc: 9.

gsl_matrix_int_free: 9.

gsl_matrix_int_get: 8, 14.

gsl_matrix_int_set: 8.

gsl_pow_2: 8.

gsl_ran_bernoulli: 8, 9.

gsl_ran_discrete: 7.

gsl_ran_discrete_free: 10.

gsl_ran_discrete_preproc: 10.

gsl_ran_discrete_t: 7, 9, 10.

gsl_ran_exponential: 8.

gsl_ran_poisson: 7.

gsl_ran_shuffle: 9.

gsl_rng: 5, 7, 8, 9, 10.

gsl_rng_alloc: 5.

gsl_rng_env_setup: 5.

gsl_rng_free: 13.

gsl_rng_mt19937: 5.

gsl_rng_set: 5.

gsl_sf_exp: 8.

gsl_sort_index: 8.

`GSL_SUCCESS`: 13.

i: 8, 9, 11, 12, 14.

indindex: 8.

iter: 9.

j: 8, 9, 14.

jindex: 9.
juvenile_genotypes: 8, 9.
k: 7, 8, 10.
L: 8, 9, 10.
leffects: 9, 13.
loceffects: 10.
locuseffects: 8.
m: 11.
main: 13.
MAX_JUVENILES: 6, 7, 9.
mean: 10.
mPop: 10.
N: 7, 8, 9, 10.
n: 12.
nruns: 10.
pindex: 9.
Pmass: 7, 10.
PmXi: 9.
Pop: 8, 9.
pow: 10.
printf: 9, 10, 14.
printmatrix: 14.
Psi: 9.
psi: 7, 10.
PXi: 10.
rep: 10.
replicates: 10, 13.
rngtype: 5, 13.
rows: 11, 14.
s: 8, 9.
seed: 5.
seleccoef: 10.
setup_rng: 5, 13.
simulator: 9, 10.
skra: 10.
svar: 10.
teljari: 7.
tempJuve: 8.
tempXi: 9.
traits: 9.
tXi: 7, 8.
xindex: 8.
Z: 8.
zbar: 9.
Zbar: 8.
znull: 8, 9, 10.

List of Refinements

⟨ Includes 15 ⟩ Used in chunk 13.

⟨ allocate array 11 ⟩ Used in chunk 13.

⟨ assign juvenile genotypes 8 ⟩ Used in chunk 13.

⟨ discrete-time simulator 9 ⟩ Used in chunk 13.

⟨ free 2d array 12 ⟩ Used in chunk 13.

⟨ initialize distribution for X_i 7 ⟩ Used in chunk 13.

⟨ many runs 10 ⟩ Used in chunk 13.

⟨ object definitions 6 ⟩ Used in chunk 13.

⟨ print matrix 14 ⟩ Used in chunk 13.

⟨ random number generator 5 ⟩ Used in chunk 13.