Fixation at two loci

Bjarki Eldon[1] [2]

Abstract

This C++ code generates excursions of the evolution of a diploid population partitioned into two genetic types at two loci, with viability weight determined by $W = e^{-s(z_0-z)^2}$, where z is the trait value of a given individual, and z_0 is the optimal trait value, and $s > 0$ is the strength of selection. The population evolves according to a model of random sweepstakes and viability selection and randomly occurring bottlenecks. We estimate the probability of fixation of the type conferring advantage, and the expected time to fixation conditional on fixation of the type conferring selective advantage at the two loci.

Contents

[1]MfN Berlin, Germany

[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 the Icelandic Centre of Research through an Icelandic Research Fund Grant of Excellence no. 185151-051 to Einar Árnason, Katrín Halldórsdóttir, Alison M. Etheridge, WS, and BE. BE also acknowledges Start-up module grants through SPP 1819 with Jere Koskela and Maite Wilke-Berenguer, and with Iulia Dahmer.
February 6, 2022

1 Copyright

2 Compilation, output and execution

This CWEB [4] document (the `.w` file) can be compiled with `cweave` to generate a `.tex` file, and with `ctangle` to generate a `.c` [3] 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. Using a Makefile can be helpful, calling this file `iguana.w`

```
iguana.pdf :  iguana.tex
     cweave iguana.w
     pdflatex iguana
     bibtex iguana
     pdflatex iguana
     pdflatex iguana
     ctangle iguana
     c++ -Wall -Wextra -pedantic -O3 -march=native -m64 iguana.c -lm -lgsl
-lgslcblas


clean :
     rm -vf iguana.c iguana.tex
```

Use `valgrind` to check for memory leaks:

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

3 introduction

We consider a diploid population of maximum size $2N$ diploid individuals. Let $X^N, X_1^N, \ldots, X_N^N$ be i.i.d. discrete random variables taking values in $\{2, \ldots, \Psi_N\}$; the $X_1^N, \ldots, X_N^N$ denote the random number of diploid juveniles independently produced in a given generation according to

$$\mathbb{P}\left(X^N = k\right) = \frac{(\Psi_N + 1)^\alpha}{(\Psi_N + 1)^\alpha - (1/2)^\alpha}\left(\frac{1}{k^\alpha} - \frac{1}{(k+1)^\alpha}\right), \quad 1 \leq k \leq \Psi_N. \tag{1}$$

The mass in Eq (1) is normalised so that $\mathbb{P}\left(2 \leq X^N \leq \Psi_N\right) = 1$, and $\mathbb{P}\left(X^N = k\right) \geq \mathbb{P}\left(X^N = k+1\right)$. Given a pool of at least N juveniles, we sample N juveniles for the next generation. Leaving out an atom at zero and one gives $X_1^N + \cdots + X_N^N \geq 2N$ almost surely, guaranteeing that we always have at least $2N$ juveniles to choose from in each generation.

Write $X_1 \sim L(\alpha, \Psi_N)$ if X_1 is distributed according to Eq (1) for given values of α and Ψ_N. Let $0 < \alpha_1 < 2$ and $\alpha_2 > 2$ be fixed and consider the mixture distribution [2]

$$X_1, \ldots, X_N \sim \begin{cases} L(\alpha_1, \Psi_N) & \text{with probability } \varepsilon_N, \\ L(\alpha_2, \Psi_N) & \text{with probability } 1 - \varepsilon_N. \end{cases} \tag{2}$$

Similarly, by identifying the appropriate scaling of ε_N one can keep α fixed and varied Ψ_N [1].

Each diploid juvenile inherits two alleles, one from each parent, and is assigned a viability weight z accoring to the two-locus type; the wild type is assigned the weight $z = e^{-sf(g)}$ for some fixed $s > 0$ and $f(g)$ is a function for how the two-locus type affects the weight. If the total number of juveniles at any given time exceeds $2N$ we sample an exponential with rate the given viability weight, and $2N$ juveniles with the smallest exponential replace the parents. In any given generation a bottleneck of a fixed size N_b occurs with a fixed probability. If a bottleneck occurs we sample N_b individuals independently and uniformly at random without replacement. The surviving individuals then produce juveniles, and if the total number of juveniles is less than the capacity $2N$ all the juveniles survive, otherwise we assign weights and sample $2N$ juveniles according to the weights as just described.

At both loci there are two types (0, 1), so there are three genotypes at each locus, and nine two-locus genotypes. Let $Y_t \equiv \{Y_t : t \geq 0\}$ denote the frequency of the two-locus type configuration in the population, i.e. Y_t takes values in $[0, 2N]^9$, and write $T_k(y) := \min\{t \geq 0 : Y_t = k, Y_0 = y\}$. We are interested in the quantitites

$$\begin{aligned} p_N(y_0) &:= \mathbb{P}\left(T_N(y_0) < T_0(y_0)\right) \\ \tau_N(y_0) &:= \mathbb{E}\left[T_N(y_0) : T_N(y_0) < T_0(y_0)\right] \end{aligned} \tag{3}$$

where y_0 is the starting configuration, i.e. the number of diploid individuals of each two-locus genotype at time 0. The quantity $p_N(y_0)$ is the probability of all $2N$ diploid individuals reaching the two-locus type conferring maximum advantage when starting from configuration y_0, and $\tau_N(y_0)$ is the expected time to do so conditional on the population reaching the configuration when starting from y_0. For example, if type 1 confers advantage at both loci, then the two-locus type in question would be the type $1/1 - 1/1$ where individuals are homozygous for type 1 at both loci.

4 Code

We collect the key containers and constants into a struct § **??**, we use the GSL random number generator § 4.2, in § **??** we compute the cumulative density function for sampling random numbers of juveniles according to the inverse CDF method, in § **??** we sample a random number of juveniles, in § **??** we define a comparison function for sorting the exponentials in § **??**, in § **??** we sample a pool of juveniles and assign weight to them in § **??**, in § **??** we sample the number of individuals of the advantageous type surviving a bottleneck, in § **??** we count the number of advantageous type surviving selection according to their weight, in § 4.27 we step through one generation by checking if a bottleneck occurs and then produce juveniles if neither fixation nor loss of the advantageous type occurs, in § 4.28 we generate one excursion until fixation or loss of the advantageous type starting with one copy of the advantageous type, the main module § 4.32 generates a given number of trajectories, § 5 holds examples of trajectories to fixation of the advantageous type.

4.1 Includes

The included libraries.

5 ⟨ includes 5 ⟩ ≡

```
#include <iostream>
#include <fstream>
#include <vector>
#include <random>
#include <functional>
#include <memory>
#include <utility>
#include <algorithm>
#include <cstddef>
#include <ctime>
#include <cstdlib>
#include <cmath>
#include <list>
#include <string>
#include <fstream>
#include <chrono>
#include <forward_list>
#include <assert.h>
#include <math.h>
#include <unistd.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include "diploid_excursions_random_bottlenecks.hpp"
```

This code is used in chunk 36.

4.2 the random number generator

6 ⟨ gslrng 6 ⟩ ≡

```
gsl_rng * 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);
}
```

This code is used in chunk 36.

4.3 the number of diploid individuals by index

return the number of diploid individuals by index. Let 0 denote the homozygous type 0/0, 1 the heterozygous type 0/1, and 2 the homozygous type 1/1 at each locus. The population is an array with indexes zero to nine with the configuration

Table 1: index

index	0	1	2	3	4	5	6	7	8
two-locus type	$(0,0)$	$(0,1)$	$(0,2)$	$(1,0)$	$(1,1)$	$(1,2)$	$(2,0)$	$(2,1)$	$(2,2)$

7 ⟨ numberbyindex 7 ⟩ ≡

```
static unsigned int number_diploid_individuals_by_index ( const std::vector <
        unsigned > &population, const size_t c_index )
{
  return population[c_index];
}
```

This code is used in chunk 36.

4.4 check if lost type

Check if lost type 1 conferring advantage at either locus; see Table 1.

8 ⟨ checklosttype 8 ⟩ ≡

```
static size_t check_if_lost_type ( const std::vector < unsigned > &population )
{     /* return 1 if lost type at either locus, otherwise 0
      */
  return ((population[1] + population[2] + population[4] + population[5] + population[7] +
      population[8] < 1) ∨ (population[3] + population[4] + population[5] + population[6] +
      population[7] + population[8] < 1) ? 1 : 0);
}
```

This code is used in chunk 36.

4.5 clear the container for juveniles

clear the container containing the juveniles; each juvenile is stored as a pair of genotype index (see Table 1), and viability weight.

9 ⟨ clearjuveniles 9 ⟩ ≡

```
static void removealljuveniles ( std::vector < std::pair < size_t , double >> &juveniles )
{
  juveniles.clear( );
  juveniles.shrink_to_fit( );
  assert(juveniles.size( ) < 1);
}
```

This code is used in chunk 36.

4.6 total number of juveniles

return the total number of juveniles

10 ⟨ totalnumberjuvs 10 ⟩ ≡

```
static size_t totalnumberjuveniles ( const std::vector < std::pair < size_t ,
    double >> &juveniles )
{
  return juveniles.size( );
}
```

This code is used in chunk 36.

4.7 add a juvenile with a given type index and weight

add a juvenile with a given viability weight and two-locus genotype index

11 ⟨ addjuv 11 ⟩ ≡

```
static void add_juvenile ( std::vector < std::pair < size_t , double >> &juveniles, const
        size_t g, const double weight )
{
  juveniles.push_back(std::make_pair(g, weight));
}
```

This code is used in chunk 36.

4.8 number of individuals with given types

see Table 1.

12 ⟨ numberwithgiventype 12 ⟩ ≡

```
unsigned int number_type ( const std::vector < unsigned > &population, const size_t
        c_lone, const size_t c_ltwo )
{
  return population[(3 * c_lone) + c_ltwo];
}
```

This code is used in chunk 36.

4.9 set population to zero

set number of individuals in the population to zero for all types

13 ⟨ setpopzero 13 ⟩ ≡

```
static void set_population_zero ( std::vector < unsigned > &population )
{
  std::fill(std::begin(population), std::end(population), 0);
  assert(std::accumulate(std::begin(population), std::end(population), 0) ≡ 0);
      /* for( size_t i = 0; i < 9 ; ++i) population[i] = 0;  */
}
```

This code is used in chunk 36.

4.10 current number of diploid individuals

return the current number of individuals in the population

14 ⟨ currentind 14 ⟩ ≡

```
static unsigned int current_number_individuals ( const std::vector <
    unsigned > &population )
{
  return std::accumulate(std::begin(population), std::end(population), 0);
    /* size_t s = 0 ; for( size_t i = 0 ; i < 9 ; ++i) s += population[i] ;  return (s); */
}
```

This code is used in chunk 36.

4.11 update the number of individuals of a given type

add or subtract by one the number of diploid individuals with a given type Table 1.

15 ⟨ updatecount 15 ⟩ ≡

```
static void update_count_type ( std::vector < unsigned > &population, const size_t
        c_type, const size_t add_subtract )
{
  assert(population[c_type] > 0);
  population[c_type] += (add_subtract < 1 ? 1 : -1);
}
```

This code is used in chunk 36.

4.12 return the most numerous type

16 ⟨ mosttype 16 ⟩ ≡

```
static size_t type_most_copies ( const std::vector < unsigned > &population )
{
  return std::distance(population.begin( ), std::max_element(population.begin( ),
      population.end( )));
}
```

This code is used in chunk 36.

4.13 all juveniles survive

all juveniles survive if the total number of juveniles produced at any given time does not exceed the carrying capacity.

17 ⟨ allsurvive 17 ⟩ ≡

```
static void update_population_all_juveniles ( const std::vector < std::pair < size_t ,
    double >> &juveniles, std::vector < unsigned > &population ) {
  /* set the population to zero § 4.9
   */
set_population_zero(population);
assert(current_number_individuals(population) < 1);
  /* add a juvenile to the population by updating the corresponding number of
   individuals with the type of the juvenile
   */
for (const auto &j:juveniles)
{
  population[std::get < 0 > (j)] += 1;
}
}
```

This code is used in chunk 36.

4.14 initialise the containers

initialise the containers for the population and the juveniles and the cumulative density functions; the initial population configuration y_0 is with $2N-2$ diploid individuals as double homozygous for the wild type allele, one individual $(0,1)$ and the other as $(1,0)$.

18 ⟨ initconts 18 ⟩ ≡

```
static void init_containers ( std::vector < unsigned > &population, std::vector <
    double > &cdf_one, std::vector < double > &cdf_two )
{
  population.clear( );
  population.assign(9, 0);
  assert(current_number_individuals(population) < 1);
  assert(population.size( ) ≡ 9);
  population[0] = GLOBAL_CONST_II − 2;
  population[1] = 1;
  population[3] = 1;
  assert(current_number_individuals(population) ≡ GLOBAL_CONST_II);
  cdf_one.clear( );
  cdf_two.clear( );
  cdf_one.reserve(GLOBAL_CONST_CUTOFF_ONE + 2);
  cdf_two.reserve(GLOBAL_CONST_CUTOFF_TWO + 2);      /* set ℙ (X^N < 2) = 0 Eq (1)
      */
  cdf_one.push_back(0.);
  cdf_one.push_back(0.);
  cdf_two.push_back(0.);
  cdf_two.push_back(0.);
  assert(cdf_one.size( ) ≡ 2);
  assert(cdf_two.size( ) ≡ 2);
}
```

This code is used in chunk 36.

4.15 initialise for a trajectory

initialise the population for a new trajectory

19 ⟨ inittraj 19 ⟩ ≡

```
static void init_for_trajectory ( std::vector < unsigned > &population )
{
  set_population_zero(population);
  assert(current_number_individuals(population) < 1);
  population[0] = GLOBAL_CONST_II - 2;
  population[1] = 1;
  population[3] = 1;
  assert(current_number_individuals(population) ≡ GLOBAL_CONST_II);
}
```

This code is used in chunk 36.

4.16 comparison function for sorting juveniles

comparison function for sorting juveniles according to viability weight

20 ⟨ fcom 20 ⟩ ≡

```
static bool comp ( const std::pair < size_t , double > a, const std::pair < size_t ,
        double > b )
{
  return (std::get < 1 > (a) < std::get < 1 > (b));
}
```

This code is used in chunk 36.

4.17 sort the juveniles

partially sort the juveniles and return the $2N$th sorted viability weight sorted in ascending order

21 ⟨ nth 21 ⟩ ≡

```
static double nthelm ( std::vector < std::pair < size_t , double >> &juveniles )
{       /* partially sort the weights using § 4.16
           */
  std::nth_element(juveniles.begin( ), juveniles.begin( ) + (GLOBAL_CONST_II − 1),
       juveniles.end( ), comp);
  return (std::get < 1 > (juveniles[GLOBAL_CONST_II − 1]));
}
```

This code is used in chunk 36.

4.18 sample juveniles according to weight

sample juveniles surviving selection by sampling according to viability weight

22 ⟨ samplejuvsweight 22 ⟩ ≡

```
static void sample_juveniles_according_to_weight ( std::vector < unsigned > &population,
        const std::vector < std::pair < size_t , double >> &juveniles, const
        double c_nth )
{
  assert(c_nth > 0.);
  set_population_zero(population);
    /* check that the population is correctly initialised § 4.9
     */
  assert(current_number_individuals(population) < 1);
  size_t j = 0;
  while (j < GLOBAL_CONST_II) {
    assert(j < GLOBAL_CONST_II);
    population[std::get < 0 > (juveniles[j])] += std::get < 1 > (juveniles[j]) ≤
        c_nth ? 1 : 0;
    ++j;
  }    /* check that we have sampled correct number of juveniles § 4.10
       */
  assert(current_number_individuals(population) ≡ GLOBAL_CONST_II);
}
```

This code is used in chunk 36.

4.19 genotype from an index

return the second genotype 0,1, or 2 from a given genotype index as in Table 1

23 ⟨ secondg 23 ⟩ ≡

```
static unsigned int second_locus_genotype_from_index(const int c_i)
{
  assert(c_i > −1);
  unsigned int x
  { }
  ;
  switch (c_i) {
  case 0:
    {
      x = 0;
      break;
    }
  case 1:
    {
      x = 1;
      break;
    }
  case 2:
    {
      x = 2;
      break;
    }
  case 3:
    {
      x = 0;
```

```
      break;
    }
  case 4:
    {
      x = 1;
      break;
    }
  case 5:
    {
      x = 2;
      break;
    }
  case 6:
    {
      x = 0;
      break;
    }
  case 7:
    {
      x = 1;
      break;
    }
  case 8:
    {
      x = 2;
      break;
    }
  default: break;
  }
```

```
    return x;
}       /* sample genotype of one parent */
        /* return the index of the genotype sampled */
```

This code is used in chunk 36.

4.20 sample a parent

return the genotype index (Table 1) of a sampled parent

24 ⟨ sampleparent 24 ⟩ ≡

```
static int sample_genotype_parent ( std::vector < unsigned > &p, gsl_rng*r )
{
  int i = 0;
  unsigned int nothers = current_number_individuals(p) − number_type(p, 0, 0);
  unsigned int x = gsl_ran_hypergeometric(r, number_diploid_individuals_by_index(p, 0),
      nothers, 1);
  while ((x < 1) ∧ (i < 7)) {
    ++i;      /* update the number of remaining individuals § 4.3
        */
    nothers −= number_diploid_individuals_by_index(p, i);
    x = gsl_ran_hypergeometric(r, number_diploid_individuals_by_index(p, i), nothers, 1);
  }
  i += (x < 1 ? 1 : 0);      /* update the number of remaining parents § 4.11
      */
  update_count_type(p, i, 1);      /* return the index of the genotype of the parent
      */      /* index is between 0 and 8
      */
  return i;
}
```

This code is used in chunk 36.

4.21 assign a genotype to juvenile

assign a single locus genotype to juvenile given single locus genotypes in parents by sampling one allele from each parent independently and uniformly at random

25 ⟨ genotypej 25 ⟩ ≡

```
static int assign_type_juvenile(const int gone, const int gtwo, gsl_rng * r)
{
  int g
  { }
  ;
  const double u = gsl_rng_uniform(r);
  switch (gone) {
  case 0:
    {
      g = (gtwo < 1 ? 0 : (gtwo < 2 ? (u < 0.5 ? 0 : 1) : 1));
      break;
    }
  case 1:
    {
      g = (gtwo < 1 ? (u < .5 ? 0 : 1) : (gtwo < 2 ? (u < 0.25 ? 0 : (u < 0.75 ? 1 : 2)) :
          (u < 0.5 ? 1 : 2)));
      break;
    }
  case 2:
    {
      g = (gtwo < 1 ? 1 : (gtwo < 2 ? (u < .5 ? 1 : 2) : 2));
      break;
    }
  default: break;
```

```
    }
    return g;
  }
```

This code is used in chunk 36.

4.22 sample random number juveniles

sample a random number of juveniles using the inverse CDF method, i.e. return

$$\min\{j \geq 2 : F(j) \geq u\} \tag{4}$$

where F is the CDF and u is a random uniform on the unit interval

26 ⟨ samplerandomjuvs 26 ⟩ ≡

```
static size_t sample_random_number_juveniles (const size_t c_twoone, const std::vector <
        double > &cdfone, const std::vector < double > &cdftwo, gsl_rng*r )
{
  const double u = gsl_rng_uniform(r);
  size_t j = 2;
  if (c_twoone < 2) {
    while (u > cdfone[j]) {
      ++j;
    }
  }
  else {
    while (u > cdftwo[j]) {
      ++j;
    }
  }
  assert(j > 1);
  return j;
}
```

This code is used in chunk 36.

4.23 compute the viability weight

compute the viability weight as $\exp(-sf(g))$ where $s \geq 0$ is the strength of selection and $f(g)$ is the genotype to phenotype map, i.e. we interpret $f(g)$ as a trait value, a phenotype, for the given two-locus genotype g. We take $f(g) = (h_1(g_1) + h_2(g_2))/2$ where h_1 and h_2 determine the contribution of the genotypes at the two loci resp.

27 ⟨ computew 27 ⟩ ≡

```
static double computeweight(const int g_one, const int g_two, gsl_rng * r)
{     /* (g_one < 1 ? 2 : (g_one < 2 ? 1 : 0));
        */     /* complete dominance, ie the wild type is recessive
        */     /* h_1(g_1) = 2𝟙_{g_1<1}
        */
  const double ggone = (g_one < 1 ? 2 : 0);     /* h_2(g_2) = 2𝟙_{g_2=0} + 𝟙_{g_2=1}
        */
  const double ggtwo = (g_two < 1 ? 2 : (g_two < 2 ? 1 : 0));     /* return a random
      exponential with rate exp(-sf(g)) used for sorting the juveniles according to
      viability weight
      */
  return (gsl_ran_exponential(r,
      1./exp((-GLOBAL_CONST_SELECTION) * pow((ggone + ggtwo)/2., 2.))));
}
```

This code is used in chunk 36.

4.24 sample a litter of juveniles

sample a litter of juveniles, i.e. a random number of juveniles with alleles, from a pair of parents with given genotypes

28 ⟨ litter 28 ⟩ ≡

```
static void add_juveniles_for_given_parent_pair ( const std::vector < double > &cdfone,
        const std::vector < double > &cdftwo, std::vector < std::pair <
        size_t , double >> &jvs, const int gone, const int gtwo, const
        size_t conetwo, gsl_rng*r )
{   /* gone and gtwo are the two-locus genotype indexes for the two
      parents Table 1
      */   /* first sample the number of juveniles produced by the
      parent pair § 4.22
      */
  const size_t numberj = sample_random_number_juveniles(conetwo,
      cdfone, cdftwo, r);
  assert(numberj > 1);
  int g_locus_one
  {}
  ;
  int g_locus_two
  {}
  ;
  for (size_t j = 0; j < numberj; ++j) {
      /* for each juvenile in the litter sample the two alleles § 4.21
       */
    g_locus_one = assign_type_juvenile((gone < 3 ? 0 : (gone < 6 ? 1 : 2)),
        (gtwo < 3 ? 0 : (gtwo < 6 ? 1 : 2)), r);
    g_locus_two =
```

```
                    assign_type_juvenile(second_locus_genotype_from_index(gone),
                    second_locus_genotype_from_index(gtwo), r);
                assert(g_locus_one ≡ 0 ∨ g_locus_one ≡ 1 ∨ g_locus_one ≡ 2);
                assert(g_locus_two ≡ 0 ∨ g_locus_two ≡ 1 ∨ g_locus_two ≡ 2);
                    /* given alleles add juvenile with viability weight § 4.23
                    */
                add_juvenile(jvs, (3 * g_locus_one) + g_locus_two,
                    computeweight(g_locus_one, g_locus_two, r));
            }
        }
```

This code is used in chunk 36.

4.25 pool of juveniles

generate a pool of juveniles for all parent pairs

29 ⟨ pool 29 ⟩ ≡

```
static void generate_pool_juveniles ( std::vector < std::pair < size_t , double >> &jvs,
        std::vector < unsigned > &p, const std::vector < double > &cdfone,
        const std::vector < double > &cdftwo, gsl_rng*r )
{   /* clear the container of juveniles § 4.5
     */
  removealljuveniles(jvs);
  int gone
  {}
  ;
  int gtwo
  {}
  ;   /* sample distribution of number of juveniles Eq (2)
     */
  const size_t conetwo = (gsl_rng_uniform(r) < GLOBAL_CONST_EPSILON ? 1 : 2);
    /* i runs over number of pairs that can be formed from the current
     number of individuals; if n the current number of individuals § 4.10 can
     produce ⌊n/2⌋ pairs of two-locus genotypes
     */
  assert(current_number_individuals(p) < GLOBAL_CONST_I + 1);
  const double currenti = current_number_individuals(p);
  for (double i = 0; i < floor(currenti/2.); ++i) {
      /* sample a parent genotype § 4.20
       */
    gone = sample_genotype_parent(p, r);
```

```
            /* sample another parent genotype § 4.20
             */
          gtwo = sample_genotype_parent(p, r);
          assert(gone > -1);
          assert(gtwo > -1);      /* given the parent genotypes add a litter § 4.24
             */
          add_juveniles_for_given_parent_pair(cdfone, cdftwo, jvs, gone, gtwo, conetwo, r);
        }
        assert(totalnumberjuveniles(jvs) ≥ static_cast⟨size_t⟩(currenti));
      }
```

This code is used in chunk 36.

4.26 bottleneck

sample diploid individuals surviving a bottleneck; we sample uniformly at random without replacement N_b diploid individuals by sampling the number of each two-locus genotype surviving a bottleneck; we therefore sample a hypergeometric by updating the relevent numbers each time

30 ⟨ bottle 30 ⟩ ≡

```
static void sample_surviving_bottleneck ( std::vector < unsigned > &p, gsl_rng*r )
{
  size_t i = 0;      /* p is the population indexed as in Table 1
      */      /* nothers is the number of individuals in the pot of the colour not being
      sampled; p[i] is the number of the colour being sampled
      */
  unsigned int nothers = current_number_individuals(p) - p[i];
  unsigned newn = gsl_ran_hypergeometric(r, p[i], nothers, GLOBAL_CONST_BOTTLENECK);
  unsigned int remaining = GLOBAL_CONST_BOTTLENECK - newn;
     /* update count of individuals of type index i surviving bottleneck
      */
  p[i] = newn;
  while ((i < 7)) {
    ++i;
    nothers -= p[i];
    newn = (remaining > 0 ? gsl_ran_hypergeometric(r, p[i], nothers, remaining) : 0);
    p[i] = newn;
    remaining -= newn;
  }      /* update for index 8 with the remaining to sample
       */
  p[8] = (remaining < GLOBAL_CONST_BOTTLENECK ? remaining : GLOBAL_CONST_II);
  assert(current_number_individuals(p) ≥ GLOBAL_CONST_BOTTLENECK);
```

```
}
```

This code is used in chunk 36.

4.27 take one step

step through one generation by first checking if a bottleneck occurs, and then sample juveniles if the type neither lost nor fixed

31 ⟨ onestep 31 ⟩ ≡

```
static void onestep ( std::vector < unsigned > &p, const std::vector < double > &cdfone,
    const std::vector < double > &cdftwo, std::vector < std::pair < size_t ,
    double >> &jvs, gsl_rng*r )
{
  double nth
  {}
  ;     /* check if bottleneck */
  if (gsl_rng_uniform(r) < GLOBAL_CONST_PROBABILITY_BOTTLENECK)
    {     /* bottleneck occurs ; sample surviving types § 4.26 and update
         population p */
    sample_surviving_bottleneck(p, r);
  }     /* first check if lost type at either loci § 4.4
       */
  if (check_if_lost_type(p) < 1) {
       /* not lost type; check if fixed at both § 4.3
       */
    if (number_diploid_individuals_by_index(p, 8) < GLOBAL_CONST_II) {
         /* not all individuals of type 2, so sample juveniles § 4.25
         */
      generate_pool_juveniles(jvs, p, cdfone, cdftwo, r);
      if (totalnumberjuveniles(jvs) ≤ GLOBAL_CONST_II) {
           /* total number of juveniles not over capacity so all survive § 4.13
           */
        update_population_all_juveniles(jvs, p);
```

```
            assert(current_number_individuals(p) ≥ GLOBAL_CONST_BOTTLENECK);
        }
        else {      /* need to sort juveniles § 4.17 and sample according to
                       weight § 4.18
                       */
            nth = nthelm(jvs);
            sample_juveniles_according_to_weight(p, jvs, nth);
            assert(current_number_individuals(p) ≥ GLOBAL_CONST_II);
        }
      }     /* mutation has fixed at both loci */
    }     /* mutation has been lost */
  }
```

This code is used in chunk 36.

4.28 trajectory

generate one trajectory by stepping through the generations one step at a time § 4.27 until either lost a type or fixed at both loci, the current optimal configuration

32 ⟨ trajectory 32 ⟩ ≡

```
static void trajectory ( std::vector < unsigned > &p, const std::vector <
        double > &cdfone, const std::vector < double > &cdftwo, std::vector
        < std::pair < size_t , double >> &jvs, const int numer, gsl_rng*r ) {
    /* initialise for a trajectory § 4.15
     */
  init_for_trajectory(p); std::vector < double > excursion_to_fixation
  { }
  ;
  int timi = 0;
  while ((check_if_lost_type(p) < 1) ∧ (current_number_individuals(p) − p[8] >
        0))
    {    /* record the number of diploid individuals homozygous 1/1 at
        both loci (sites) over current number of diploid individuals */
    assert(current_number_individuals(p) ≥ GLOBAL_CONST_BOTTLENECK);
    excursion_to_fixation.push_back(static_cast⟨double⟩(number_diploid_individuals_by_in
        8))/static_cast⟨double⟩(current_number_individuals(p)));
    ++timi;
    onestep(p, cdfone, cdftwo, jvs, r);
  }
  const std::string eskra = "twolocitrajectory" + std::to_string(numer) +
      ".txt";    /* check if lost the type after an excursion § 4.4
      */
  if (check_if_lost_type(p) < 1) {    /* did not lose the type; check that
      fixed at both loci for the type conferring selective advantage § 4.12
```

```
      */
  assert(type_most_copies(p) ≡ 8);
      /* fixation occurs so print excursion to file
      */
  std::cout ≪ eskra ≪ '\n';
  std::ofstream f(eskra, std::ofstream::app);
  assert(f.is_open()); for (const auto &y:excursion_to_fixation)
  {
    f ≪ y ≪ '␣';
  }
  f ≪ '\n';
  f.close(); } std::cout ≪ (check_if_lost_type(p) > 0 ? 0 : 1) ≪ '␣' ≪ timi ≪
      '\n'; }
```

This code is used in chunk 36.

4.29 the mass function Eq (1)

The mass function in Eq (1) used for sampling random number of juveniles; see § 4.30

33 ⟨ pxn 33 ⟩ ≡

```
static double px(const double k, const double calpha, const double ccutoff)
{
  return ((pow(1./k, calpha) - pow(1./(k + 1.), calpha))/(pow(.5,
      calpha) - pow(1./(ccutoff + 1.), calpha)));
}
```

This code is used in chunk 36.

4.30 initialise the CDF from Eq (1)

Initialise the CDF for the distribution of random number of juveniles Eq (1) § 4.29; the population evolves according to Eq (2) so need two versions of the CDF for different values of α or the cutoff Ψ_N

34 ⟨ initcdf 34 ⟩ ≡

```
static void initialise_cdf ( std::vector < double > &cdfo, std::vector < double > &cdft )
{
  for (double i = 2; i ≤ GLOBAL_CONST_PSI_ONE; ++i) {
    cdfo.push_back(cdfo.back( ) + px(i, GLOBAL_CONST_ALPHA_ONE,
        GLOBAL_CONST_PSI_ONE));
  }
  for (double j = 2; j ≤ GLOBAL_CONST_PSI_TWO; ++j) {
    cdft.push_back(cdft.back( ) + px(j, GLOBAL_CONST_ALPHA_TWO,
        GLOBAL_CONST_PSI_TWO));
  }
}
```

This code is used in chunk 36.

4.31 generate a given number of trajectories

generate a given number of trajectories

35 ⟨ runsims 35 ⟩ ≡

```
static void runsims(const int cnumer, gsl_rng * r) { std::vector < unsigned
        int > population(9, 0); std::vector < std::pair < size_t , double >> juveniles
{}
; std::vector < double > cdf_one
{}
; std::vector < double > cdf_two
{}
;   /* initialise the main containers for the objects we need § 4.14
    */
init_containers(population, cdf_one, cdf_two);   /* initialise both CDFs § 4.30
    */
initialise_cdf (cdf_one, cdf_two);
int z = GLOBAL_CONST_NUMBER_EXPERIMENTS + 1;
while (--z > 0) {   /* sample a trajectory § 4.28
      */
  trajectory(population, cdf_one, cdf_two, juveniles, cnumer, r);
}
}
```

This code is used in chunk 36.

4.32 the main module

The *main* function

36 ⟨ includes 5 ⟩

⟨ gslrng 6 ⟩

⟨ numberbyindex 7 ⟩

⟨ checklosttype 8 ⟩

⟨ clearjuveniles 9 ⟩

⟨ totalnumberjuvs 10 ⟩

⟨ addjuv 11 ⟩

⟨ numberwithgiventype 12 ⟩

⟨ setpopzero 13 ⟩

⟨ currentind 14 ⟩

⟨ updatecount 15 ⟩

⟨ mosttype 16 ⟩

⟨ allsurvive 17 ⟩

⟨ initconts 18 ⟩

⟨ inittraj 19 ⟩

⟨ fcom 20 ⟩

⟨ nth 21 ⟩

⟨ samplejuvsweight 22 ⟩

⟨ secondg 23 ⟩

⟨ sampleparent 24 ⟩

⟨ genotypej 25 ⟩

⟨ samplerandomjuvs 26 ⟩

⟨ computew 27 ⟩

⟨ litter 28 ⟩

⟨ pool 29 ⟩

⟨ bottle 30 ⟩

⟨ onestep 31 ⟩

⟨ trajectory 32 ⟩

⟨ pxn 33 ⟩

⟨ initcdf 34 ⟩

⟨ runsims 35 ⟩

```
int main(int argc, char *argv[])
{
  setup_rng(static_cast⟨unsigned long int⟩(atoi(argv[1])));
      /* run a given number of trajectories § 4.31
       */
  runsims(atoi(argv[1]), rngtype);      /* free the random number generator in § 4.2
       */
  gsl_rng_free(rngtype);
  return 0;
}
```

5 examples

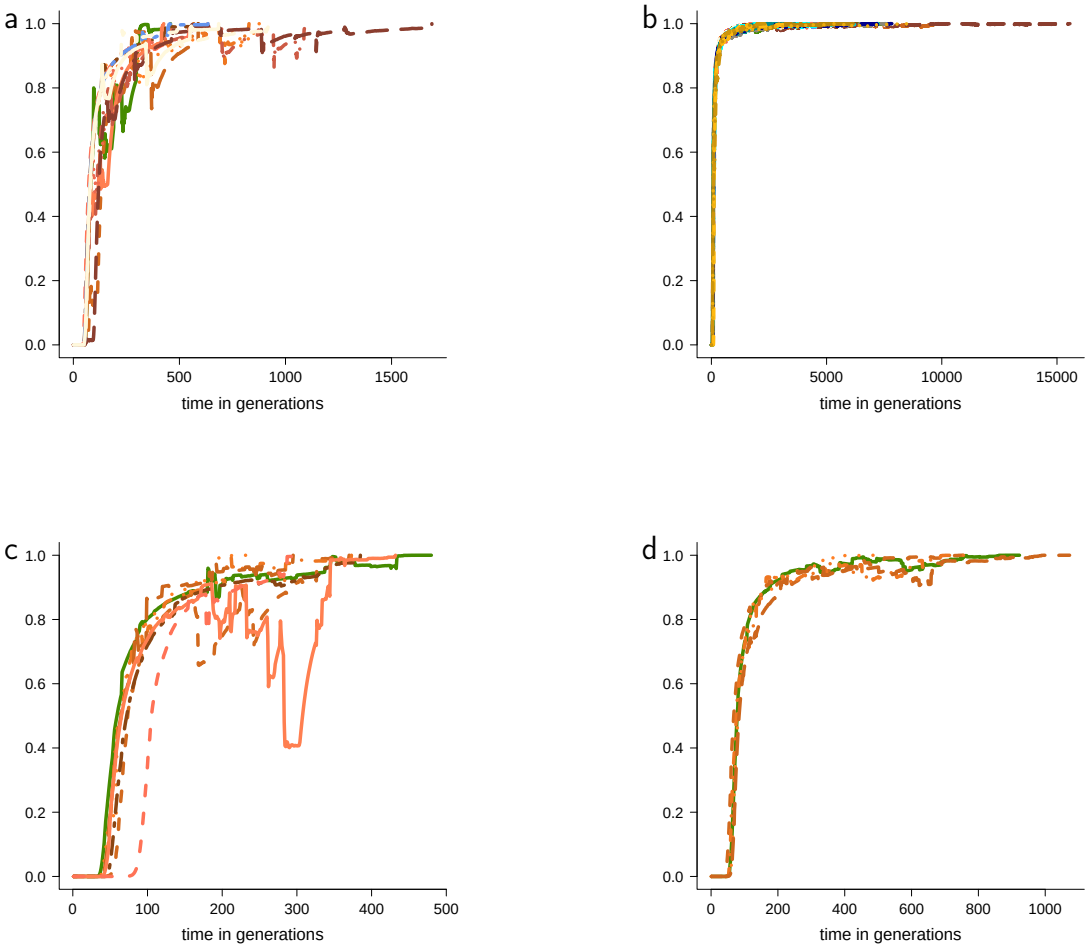

Figure 1: $2N = 10^6$, $\Psi_N = 2N$, $\alpha_2 = 3$, $\varepsilon_N = 0$ (a,b) resp. 0.1; $s = 0.5$, bottleneck size 10^2 (a,c) resp. 10^4, probability of a bottleneck in any given generation 0.01; $h_i(g) = 2\mathbb{1}_{\{g<1\}}$ ie the wild type is recessive, and viability weight $\exp\left(-s\left(\left(h_1(g_1) + h_2(g_2)\right)/2\right)^2\right)$ from 10^2 replicates. The excursions are shown as the frequency at time t of diploid individual homozygous for the beneficial type at both loci relative to the number of diploid individuals in the population at time t.

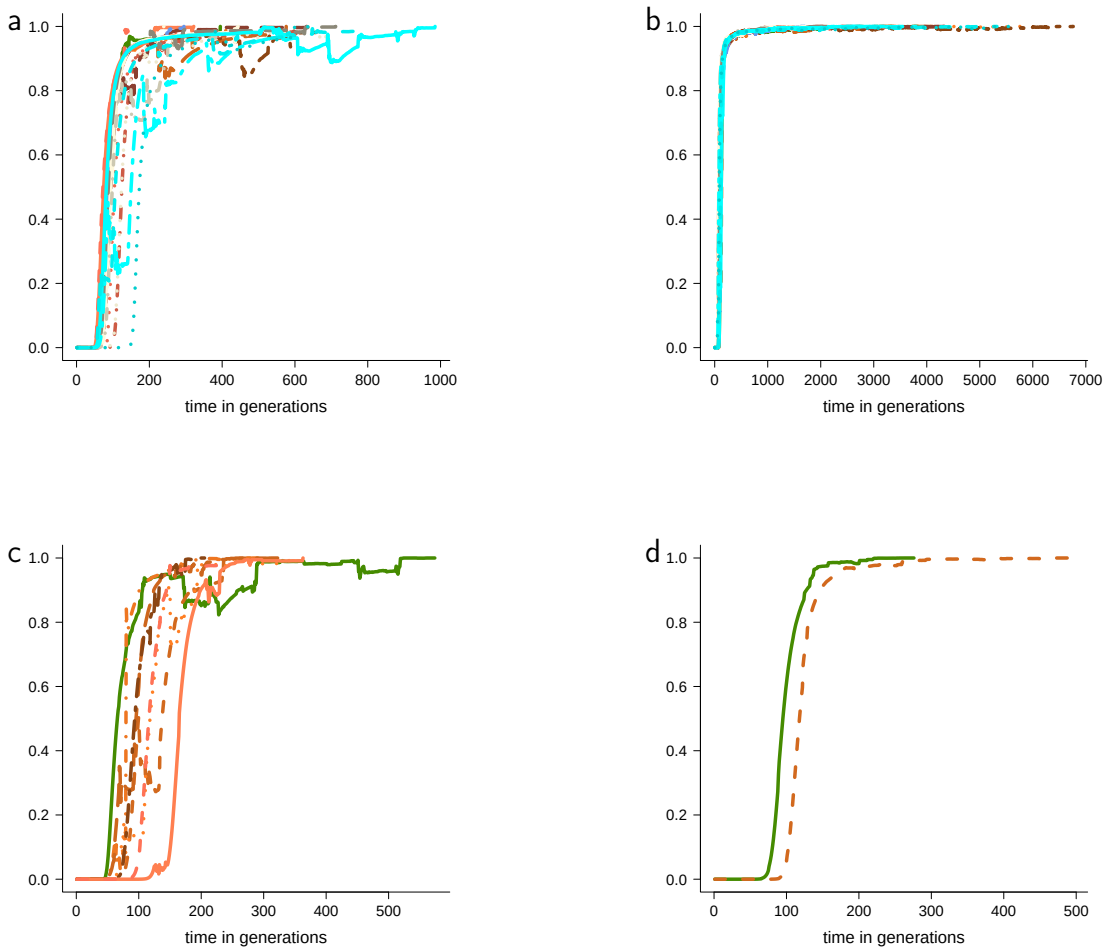

Figure 2: $2N = 10^6$, $\Psi_N = 2N$, $\alpha_2 = 3$, $\varepsilon_N = 0$ (a,b) resp. 0.1; $s = 0.5$, bottleneck size 10^2 (a,c) resp. 10^4, probability of a bottleneck in any given generation 0.01; $h_1(g) = 2\mathbb{1}_{\{g<1\}}$ ie the wild type is recessive, $h_2(g) = 2\mathbb{1}_{\{g=0\}} + \mathbb{1}_{\{g=1\}}$ and viability weight $\exp\left(-s\left(\left(h_1(g_1) + h_2(g_2)\right)/2\right)^2\right)$ from 10^2 replicates. The excursions are shown as the frequency at time t of diploid individual homozygous for the beneficial type at both loci relative to the number of diploid individuals in the population at time t.

6 conclusion

7 references

References

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

[2] Iulia Dahmer and Bjarki Eldon. Coalescent processes from models of random sweepstakes. in preparation.

[3] Brian W Kernighan and Dennis M Ritchie. The C programming language, 1988.

[4] Donald Ervin Knuth and Silvio Levy. *The CWEB system of structured documentation: version 3.0*. Addison-Wesley Longman Publishing Co., Inc., Reading, Massachusetts, 1994.

Index

List of Refinements

⟨ addjuv 11 ⟩ Used in chunk 36.

⟨ allsurvive 17 ⟩ Used in chunk 36.

⟨ bottle 30 ⟩ Used in chunk 36.

⟨ checklosttype 8 ⟩ Used in chunk 36.

⟨ clearjuveniles 9 ⟩ Used in chunk 36.

⟨ computew 27 ⟩ Used in chunk 36.

⟨ currentind 14 ⟩ Used in chunk 36.

⟨ fcom 20 ⟩ Used in chunk 36.

⟨ genotypej 25 ⟩ Used in chunk 36.

⟨ gslrng 6 ⟩ Used in chunk 36.

⟨ includes 5 ⟩ Used in chunk 36.

⟨ initcdf 34 ⟩ Used in chunk 36.

⟨ initconts 18 ⟩ Used in chunk 36.

⟨ inittraj 19 ⟩ Used in chunk 36.

⟨ litter 28 ⟩ Used in chunk 36.

⟨ mosttype 16 ⟩ Used in chunk 36.

⟨ nth 21 ⟩ Used in chunk 36.

⟨ numberbyindex 7 ⟩ Used in chunk 36.

⟨ numberwithgiventype 12 ⟩ Used in chunk 36.

⟨ onestep 31 ⟩ Used in chunk 36.

⟨ pool 29 ⟩ Used in chunk 36.

⟨ pxn 33 ⟩ Used in chunk 36.

⟨ runsims 35 ⟩ Used in chunk 36.

⟨ samplejuvsweight 22 ⟩ Used in chunk 36.

⟨ sampleparent 24 ⟩ Used in chunk 36.

⟨ samplerandomjuvs 26 ⟩ Used in chunk 36.

⟨ secondg 23 ⟩ Used in chunk 36.

⟨ setpopzero 13 ⟩ Used in chunk 36.

⟨ totalnumberjuvs 10 ⟩ Used in chunk 36.

⟨ trajectory 32 ⟩ Used in chunk 36.

⟨ updatecount 15 ⟩ Used in chunk 36.