Fixation at linked sites

Bjarki Eldon[1]

Abstract

A simulator for the evolution of a diploid population evolving according to random sweepstakes, recurrent bottlenecks, and viability selection acting on linked sites. At each site there are two types, the wild and the fit type, with the latter conferring selective advantage. We record excursions to fixation of the fit type jointly at all the sites under all kinds of dominance and epistatic mechanisms.

Contents

[1]MfN Berlin, Germany

Supported by Deutsche Forschungsgemeinschaft (DFG) - Projektnummer 273887127 through DFG SPP 1819: Rapid Evolutionary Adaptation grant STE 325/17 to Wolfgang Stephan; BE acknowledges 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 M. Etheridge, and BE; and SPP 1819 Start-up module grants with Jere Koskela and Maite Wilke Berenguer, and with Iulia Dahmer
October 25, 2022

1 Copyright

Copyright © 2022 Bjarki Eldon

This document and any source code it contains is distributed under the terms of the GNU General Public Licence (version ≥ 3). You should have received a copy of the licence along with this file (see file COPYING).

The source codes described in this document are free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This document and the code it contains is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this file (see COPYING). If not, see `http://www.gnu.org/licenses/`.

2 intro

?⟨SEC:intro⟩?

Consider a diploid population evolving in discrete (non-overlapping) generations according to random sweepstakes and recurrent bottlenecks. In each generation the current individuals randomly form pairs, and each pair independently produces a random number of potential offspring (juveniles) according to a given law. If the total number of juveniles so produced exceeds a fixed carrying capacity $\mathfrak{C}$ we sample $\mathfrak{C}$ of them without replacement based on their viability weight, otherwise all the juveniles survive.

Let $X, X_1, \ldots, X_M$ for $M \in \mathbb{N} := \{1, 2, \ldots\}$ denote iid copies of positive random variables with

$$\mathbb{P}(X = k) = \mathbb{1}_{\{2 \le k \le u(N)\}} C \left(\frac{1}{k^\alpha} - \frac{1}{(k+1)^\alpha} \right) \tag{1}$$ eq:1

where $\alpha, C > 0$ are constants so that

$$\mathbb{P}(X = k) \ge \mathbb{P}(X = k+1), \quad k \ge 0;$$
$$\mathbb{P}(2 \le X \le u(N)) = 1$$

We choose the law so that $\mathbb{P}(X \ge 2M) = 1$.

We randomize on α in Eq (1). Fix $0 < \alpha_1 < 2$ and $\alpha_2 \ge 2$. With probability ε all current parent pairs independently produce juveniles with $\alpha = \alpha_1$, and with probability $1 - \varepsilon$ we have $\alpha = \alpha_2$. Such a mixture can be shown to have better properties than fixing α.

Recurrent bottlenecks are modelled by tossing a coin at the start of a given generation; if a bottleneck occurs we sample a fixed number B of diploid individuals to survive a bottleneck and produce juveniles who all survive if the total number of juveniles does not exceed the carrying capacity $\mathfrak{C}$. Let N_t denote the population size, the number of diploid individuals at time t. Then

$$N_{t+1} = \mathfrak{C}\mathbb{1}_{\{S_{\lfloor M/2 \rfloor} > \mathfrak{C}\}} + S_M \mathbb{1}_{\{S_{\lfloor M/2 \rfloor} \le \mathfrak{C}\}} \tag{2}$$?eq:2?

where S_N denotes the total number of juveniles produced by N parent pairs and

$$M = B\mathbb{1}_{\{\text{bottleneck occurs}\}} + N_t\mathbb{1}_{\{\text{bottleneck does not occur}\}} \tag{3}$$?eq:3?

When the total number of juveniles exceeds $\mathfrak{C}$ we sample for each juvenile a random exponential with rate the viability weight of the juvenile. The juveniles with the $\mathfrak{C}$ smallest exponentials survive. This is a way of applying viability selection to the evolution of the population. The viability weight of each juvenile is determined by the genotypes at all the sites.

2.1 pseudocode

c:pseudocode)?

3 code

3.1 the included libraries

⟨sec:includes⟩

the included libraries and the header file

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 <bitset>
#include <unistd.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
```

```
#include <gsl/gsl_sf.h>
#include "Lsitesalllinked.hpp"
```

This code is used in chunk 30.

3.2 GSL random number generator

⟨sec:gslrng⟩

initialise a GSL random number generator; see § 3.26

6 ⟨ gsl random number generator 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 30.

3.3 lookup table

⟨sec:lookup⟩

generate a lookup table where entry with index i contains the corresponding haplotype indexes; see § 3.25

7 ⟨generate lookup table 7⟩ ≡

```
static void generate_lookup_table ( std::vector < std::pair < unsigned long , unsigned
    long >> &table )
{
  table.clear( );
  table.shrink_to_fit( );
  assert(table.size( ) < 1);
  for (unsigned long i = 0; i ≤ GLOBAL_CONST_MAX_INDEX; ++i) {
    for (unsigned long j = i; j ≤ GLOBAL_CONST_MAX_INDEX; ++j) {
      table.push_back(std::make_pair(i, j));
    }
  }
}
```

This code is used in chunk 30.

3.4 the lookup function

ookupfunction⟩

going along rows in the array of number of individuals of each diploid phased L-site type; see § 3.8

8 ⟨ lookup function 8 ⟩ ≡

```
static unsigned long lookup(const unsigned long i, const unsigned long j)
{     /* i and j with i ≤ j are the two haplotype indexes
        */     /* GLOBAL_CONST_MAX_INDEX is the index of haplotype 1...1
        */
  return (j + (i * GLOBAL_CONST_MAX_INDEX) − (i * (i − 1)/2));
}
```

This code is used in chunk 30.

3.5 check if recombination occurs

recombination⟩

check if parent haplotypes recombine and produce recombinant haplotypes; we check this regardless of the configuration of the haplotypes, i.e. if the recombinants would be different from the originals or not. If a random uniform does not exceed $r(L-1)$ where r is the recombination probability between a pair of sites recombination occurs

9 ⟨ check for recombination 9 ⟩ ≡

```
static int recombination()
{     /* if recombination return a discrete uniform x between 1 and L − 1
        */     /* recombination then occurs between x − 1 and x
        */
  return (gsl_rng_uniform(rngtype) ≤ (GLOBAL_CONST_RECOMBINATION *
        (GLOBAL_CONST_NUMBER_SITESd − 1)) ? gsl_rng_uniform_int(rngtype,
        GLOBAL_CONST_NUMBER_SITES − 1) + 1 : GLOBAL_CONST_NUMBER_SITES + 2);
       /* gsl_rng_uniform_int(rngtype, n) returns a random int between zero and n − 1
        */     /* shifting by one returns an integer between one and L − 1
        */
}
```

This code is used in chunk 30.

3.6 generate recombinant haplotypes

anthaplotypes⟩

generate recombinant haplotypes given recombination occurs; return one of the two recombined haplotypes picked with equal probability; called in § 3.7;

if the parent haplotypes are $(a_1, a_2, \ldots, a_L)$ and $(b_1, \ldots, b_L)$ and recombination occurs between sites $\ell-1$ and ℓ then the two recombinant haplotypes are $(a_1, \ldots, a_{\ell-1}, b_\ell, b_{\ell+1}, \ldots, b_L)$ and $(b_1, \ldots, b_{\ell-1}, a_\ell, a_{\ell+1}, \ldots, a_L)$

10 ⟨ recombine haplotypes 10 ⟩ ≡

```
static unsigned long recombine(const int lrec, const unsigned long indexhapone, const
        unsigned long indexhaptwo)
{   /* indexhapone and indexhaptwo are the indexes for the two parent haplotypes
       */   /* recombination happens between site lrec − 1 and lrec
       */   /* for lrec between one and L − 1; L is number of sites
       */
  std::stringhapone = std::bitset < GLOBAL_CONST_NUMBER_SITES >
      (indexhapone).to_string();
  std::stringhaptwo = std::bitset < GLOBAL_CONST_NUMBER_SITES >
      (indexhaptwo).to_string();
  std::stringtmp = hapone;
     /* generate the recombinat haplotypes by replacing the respective ends
       */
  hapone.replace(lrec, GLOBAL_CONST_NUMBER_SITES − lrec, haptwo.substr(lrec,
      GLOBAL_CONST_NUMBER_SITES − lrec));
  haptwo.replace(lrec, GLOBAL_CONST_NUMBER_SITES − lrec, tmp.substr(lrec,
      GLOBAL_CONST_NUMBER_SITES − lrec));
     /* return the index of the recombinant haplotype picked for the juvenile
       */
  return (gsl_rng_uniform(rngtype) < 0.5 ? std::bitset < GLOBAL_CONST_NUMBER_SITES >
      (hapone).to_ulong() : std::bitset < GLOBAL_CONST_NUMBER_SITES >
```

```
        (haptwo).to_ulong());
}
```

This code is used in chunk 30.

3.7 sample a haplotype index

aplotypeindex⟩

sample haplotype index from a parent given haplotype indexes of the parent; the index may represent a recombinant

11 ⟨ sample haplotype index 11 ⟩ ≡

```
static unsigned long sample_haplotype_index(const unsigned long hone, const
        unsigned long htwo)
{   /* hone and htwo are the haplotype indexes of the two parents;
    */   /* check if recombination occurs; see § 3.5 for recombination
    */
  const int x = recombination( );
    /* if x < number of sites then recombination occurs
    */   /* otherwise return the index of one or the other of the parent haplotypes;
    see § 3.6 for recombine
    */
  return (x < GLOBAL_CONST_NUMBER_SITES ? recombine(x, hone,
    htwo) : (gsl_rng_uniform(rngtype) < 0.5 ? hone : htwo));
}
```

This code is used in chunk 30.

3.8 initializing the population array

sec:initarray⟩

initializing the array of number of diploid individuals of each phased L-site type

12 ⟨ initialize population array 12 ⟩ ≡

```
static void initializearray ( std::vector < unsigned long > &p )
{    /* p is the population; each entry is the number of individuals of the respective
        phased L-site type
        */
  std::strings(GLOBAL_CONST_NUMBER_SITES, '0');
  std::fill(std::begin(p), std::end(p), 0);
  assert(std::accumulate(std::begin(p), std::end(p), 0) < 1);
  for (int i = 0; i < GLOBAL_CONST_NUMBER_SITES; ++i) {
    s[i] = '1';    /*
        § 3.4 for lookup */
    p[lookup(0, std::bitset < GLOBAL_CONST_NUMBER_SITES > (s).to_ulong())] = 1;
    s[i] = '0';
  }    /* L number of diploid individuals carry a fit type at one site, i.e. are
        heterozygous at one site only
        */    /* and at all other sites homozygous for the wild type;
        */
       /* all other diploid individuals are homozygous for the wild type at all sites
        */
  p[0] = GLOBAL_CONST_CARRYING_CAPACITY − GLOBAL_CONST_NUMBER_SITES;
}
```

This code is used in chunk 30.

3.9 current number of diploid individuals

⟨sec:Nt⟩

compute the current number of diploid individuals

13 ⟨ current number *Nt* 13 ⟩ ≡

```
static unsigned long current_number_individuals ( const std::vector < unsigned
    long > &p )
{
  return std::accumulate(std::begin(p), std::end(p), 0);
}
```

This code is used in chunk 30.

3.10 viability weight

abilityweight⟩

compute viability weight of a juvenile given haplotype indexes

14 ⟨ viability weight 14 ⟩ ≡

```
static double weight ( const std::vector < unsigned long > &h )
{    /* h contains haplotype indexes of juvenile
        */    /* convert the indexes to a string of zeros and ones; zero for wild type
        and one for fit type
        */
  const std::string hapone = std::bitset < GLOBAL_CONST_NUMBER_SITES >
      (h[0]).to_string();
  const std::string haptwo = std::bitset < GLOBAL_CONST_NUMBER_SITES >
      (h[1]).to_string();
  double w = 0;    /* need to be homozygous for fit type to increase weight by one
      */    /* (g1, g2) ↦ 𝟙{g1=2} + 𝟙{g2=2} and no epistasis
      */
  for (int i = 0; i < GLOBAL_CONST_NUMBER_SITES; ++i) {
    w += (hapone[i] ≡ '1' ? (haptwo[i] ≡ '1' ? 1. : 0) : 0.);
  }    /* adding a small deviation not necessary : gsl_ran_gaussian_ziggurat(rngtype, 0.1)
        */
  return (gsl_ran_exponential(rngtype, 1./(1. + (GLOBAL_CONST_SELECTION * w))));
}
```

This code is used in chunk 30.

3.11 probability kernel for number of juveniles

⟨sec:kernel⟩

compute a kernel according to Eq (1) for sampling a random number of juveniles

15 ⟨ kernel 15 ⟩ ≡

```
static double masskernel(const double k, const double alpha)
{
  return (pow(1./k, alpha) - pow(1./(k + 1.), alpha));
}
```

This code is used in chunk 30.

3.12 the CDF for distribution of number of juveniles

⟨sec:cdf⟩

generate the CDF for sampling random number of juveniles

16 ⟨ cdf 16 ⟩ ≡

```
static void cdf_number_juveniles ( std::vector < double > &x, const double a )
    {   /* a is α in Eq (1)
         */
      for (unsigned long k = 2; k ≤ GLOBAL_CONST_CARRYING_CAPACITY; ++k) {
          /*
           § 3.11 */
        x[k] = (masskernel(static_cast⟨double⟩(k), a));
      }
      const double cconst = std::accumulate(std::begin(x), std::end(x), 0.);
      for (unsigned long k = 2; k ≤ GLOBAL_CONST_CARRYING_CAPACITY; ++k) {
        x[k] = x[k − 1] + (masskernel(static_cast⟨double⟩(k), a)/cconst);
      }   /* the cdf must have last element one to guarantee a value within limits is
             sampled
             */
      x.back( ) = 1.;
    }
```

This code is used in chunk 30.

3.13 litter size

ec:littersize⟩

sample a random number of juveniles for one family

17 ⟨ sample litter size 17 ⟩ ≡

```
static int sample_litter_size ( const std::vector < double > &x )
{
  int j = 2;
  const double u = gsl_rng_uniform(rngtype);
      /* with F denoting the CDF return min{j ∈ {2, 3, ..., u(N)} : u ≤ F(j)}
       */
  while (u > x[j]) {
    ++j;
  }
  assert(j > 1);
  assert(j ≤ static_cast⟨int⟩(GLOBAL_CONST_CUTOFF));
  return j;
}
```

This code is used in chunk 30.

3.14 adding juvenile

sec:addingjuv⟩

add a juvenile to the litter

18 ⟨ addjuv 18 ⟩ ≡

```
static void add_juvenile ( const std::vector < unsigned long > &pone, const std::vector
        < unsigned long > &ptwo, std::vector < std::pair < std::vector < unsigned
        long > , double >> &vj, std::vector < double > &v_juvenileweights ) {
      /* pone and ptwo are the pairs of haplotype indexes for the two parents
        */
    std::vector < unsigned long > h
    { }
    ;    /* sample haplotype from parent one (arbitrarily enumerated) with haplotype
        indexes in pone
        */
    h.push_back(sample_haplotype_index(pone[0], pone[1]));    /* sample haplotype
        from parent two (aribtrarily enumerated) with haplotype indexes in ptwo
        */
    h.push_back(sample_haplotype_index(ptwo[0], ptwo[1]));
      /* h now contains the indexes of the two haplotypes carried by the juvenile; the
        genome of the juvenile
        */    /* compute the weight of the juvenile given the indexes § 3.10
        */
    const double w = weight(h);
      /* record the weight of the juvenile for computing the nth smallest
        */
    v_juvenileweights.push_back(w);
    vj.push_back(std::make_pair(h, w)); }
```

This code is used in chunk 30.

3.15 add sibship to pool of juveniles

⟨sec:sibship⟩

add litter or sibship to pool of juveniles

19 ⟨ add sibship 19 ⟩ ≡

```
static void add_litter ( const std::vector < double > &v_cdf, const std::vector <
        unsigned long > &po, const std::vector < unsigned long > &pt,
        std::vector < std::pair < std::vector < unsigned long > , double >> &pool,
        std::vector < double > &vw )
{   /* sample litter size § 3.13
     */
  const int littersize = sample_litter_size(v_cdf);
  for (int j = 0; j < littersize; ++j) {
      /* given sibship size add juveniles one by one § 3.14
       */
    add_juvenile(po, pt, pool, vw);
  }
}
```

This code is used in chunk 30.

3.16 sample one diploid individual

oneindividual⟩

sample one index of phased L-site type of diploid parent and update the population; used in § 3.24 for removing individuals not surviving a bottleneck

20 ⟨ onehypergeometric 20 ⟩ ≡

```
static unsigned long sample_genotype_parent ( std::vector < unsigned long > &p )
{     /* p is population
        */
  unsigned long i = 0;
  unsigned int nothers = static_cast⟨unsigned
      int⟩(current_number_individuals(p) − p[i]);
  unsigned int x = gsl_ran_hypergeometric(rngtype, p[0], nothers, 1);
  while ((x < 1) ∧ (i < GLOBAL_CONST_TOTAL_NUMBER_PHASED_TYPES)) {
    ++i;
    nothers −= p[i];
    x = gsl_ran_hypergeometric(rngtype, p[i], nothers, 1);
  }     /* check if an individual has been sampled
          */
  i += (x < 1 ? 1 : 0);     /* adjust the number of remaining parents
      */     /* an individual of type with index i sampled, so subtract one from the
      number of remaining individuals with same type
      */
  −p[i];     /* return the sampled index of the diploid phased L-site type of the parent
      */
     /* index is between zero and GLOBAL_CONST_TOTAL_NUMBER_PHASED_TYPES
      */
  return i;
}
```

This code is used in chunk 30.

3.17 clear container for juveniles

sec:clearpool⟩

clear the container for the pool of juveniles; may not need this but is here nevertheless

21 ⟨ clear the pool 21 ⟩ ≡

```
static void clearpooljuveniles ( std::vector < std::pair < std::vector < unsigned long > ,
    double >> &x ) { for (auto &y:x) { std::get < 0 > (y).clear();
std::get < 0 > (y).shrink_to_fit(); std::vector < unsigned
    long > ().swap(std::get < 0 > (y)); } x.clear();
x.shrink_to_fit(); std::vector < std::pair < std::vector < unsigned long > ,
    double >> ().swap(x);
assert(x.size() < 1); }
```

This code is used in chunk 30.

3.18 a new pool of juveniles

⟨sec:newpool⟩

generate a new pool of juveniles

22 ⟨ a new pool 22 ⟩ ≡

```
static void new_pool_juveniles (const unsigned long Nt,
        std::vector < std::pair < std::vector < unsigned long > , double >> &pool,
        std::vector < double > &vw, std::vector < unsigned long > &population,
        const std::vector < double > &vcdf, const std::vector < std::pair <
        unsigned long , unsigned long >> &table ) {
      /* Nt is current number of individuals
        */    /* pool is the vector for the new pool of juveniles § 3.17
        */
    clearpooljuveniles(pool);
    vw.clear( );
    vw.shrink_to_fit( );
    assert(vw.size( ) < 1);
    unsigned long indexpone
    { }
    ;
    unsigned long indexptwo
    { }
    ; std::vector < unsigned long > pone(2, 0); std::vector < unsigned
        long > ptwo(2, 0);    /* Nt is current number of individuals
        */    /* each time sample two parents
        */
    for (unsigned long i = 0; i < Nt/2; ++i) {
        /* sample index of a phased L-site type of parent one
         */
        /* table[indexpone] contains the corresponding haplotype indexes § 3.16
```

```
        */
    indexpone = sample_genotype_parent(population);
    pone[0] = std::get < 0 > (table[indexpone]);
    pone[1] = std::get < 1 > (table[indexpone]);
        /* table[indexptwo] contains the corresponding haplotype indexes
        */
    indexptwo = sample_genotype_parent(population);
    ptwo[0] = std::get < 0 > (table[indexptwo]);
    ptwo[1] = std::get < 1 > (table[indexptwo]);
        /* given parent genotypes sample a sibship
        § 3.15 */
    add_litter(vcdf, pone, ptwo, pool, vw);
  }
}
```

This code is used in chunk 30.

3.19 count homozygous

ec:counthomoz⟩

given a site for which to check, count how many diploid individuals are homozygous for the wild type at the site

23 ⟨ count homozygous at site 23 ⟩ ≡

```
static unsigned long checknullsite (const unsigned long nsite, const std::vector <
        unsigned long > &population, const std::vector < std::pair < unsigned long
        , unsigned long >> &tafla )
{   /* nsite is the site of interest; we record how many individuals are homozygous
        for the wild type at site nsite
        */
  unsigned long sumof = 0;
  for (unsigned long i = 0; i < GLOBAL_CONST_TOTAL_NUMBER_PHASED_TYPES;
        ++i) {   /* if the type is homozygous for the wild type at site nsite add
        the number of individuals with the phased L-site type
        */
    sumof += (std::bitset < GLOBAL_CONST_NUMBER_SITES >
        (std::get < 0 > (tafla[i])).to_string()[nsite] == '0' ?
        (std::bitset < GLOBAL_CONST_NUMBER_SITES > (std::get < 1 >
        (tafla[i])).to_string()[nsite] == '0' ? population[i] : 0) : 0);
  }   /* return the number of individuals homozygous for the wild type at site
        nsite
        */
  return sumof;
}
```

This code is used in chunk 30.

3.20 compare two values

⟨sec:comp⟩

compare two values for computing the nth smallest element § 3.21

24 ⟨ compare 24 ⟩ ≡

```
static bool comp(const double a, const double b)
{
  return a < b;
}
```

This code is used in chunk 30.

3.21 *n*th smallest element

⟨sec:nth⟩

compute the *n*th element for sampling juveniles when the total number of juveniles exceeds the carrying capacity

25 ⟨ *n*th element 25 ⟩ ≡

```
static double nthelm ( std::vector < double > &weights )
{      /* § 3.20
          */
  std::nth_element(weights.begin( ),
        weights.begin( ) + (GLOBAL_CONST_CARRYING_CAPACITY − 1), weights.end( ), comp);
  return weights[GLOBAL_CONST_CARRYING_CAPACITY − 1];
}
```

This code is used in chunk 30.

3.22 sample juveniles

sortjuveniles⟩

sort juveniles and update population

26 ⟨ sorting 26 ⟩ ≡

```
static void select_juveniles_according_to_weight ( std::vector < double > &v_weights,
        std::vector < unsigned long > &v_population, const std::vector <
        std::pair < std::vector < unsigned long > , double >> &v_pool ) {
      /* compute the nth smallest viability weight
        */    /* only sample juveniles according to weight if the total number of
        juveniles exceeds the carrying capacity
        */    /* otherwise all juveniles survive
        */
    const double wnth = (static_cast<unsigned long>(v_pool.size()) >
        GLOBAL_CONST_CARRYING_CAPACITY ? nthelm(v_weights) :
        std::max_element(std::begin(v_weights), std::end(v_weights))[0] + 1.);
    assert(wnth > 0.);    /* set number of all phased L-site types to zero
        */
    std::fill(std::begin(v_population), std::end(v_population), 0);
    assert(std::accumulate(std::begin(v_population), std::end(v_population), 0.) < 1);
      /* add to count of phased L-site type of juvenile if juvenile survives
        */
    for (const auto &j:v_pool)
    {    /*
          see § 3.4 for lookup */
      v_population[lookup(std::get < 0 > (j)[0],
          std::get < 0 > (j)[1])] += (std::get < 1 > (j) ≤ wnth ? 1 : 0);
    }
}
```

This code is used in chunk 30.

3.23 check if lost a fit type

checklosttype⟩

count the number of individuals homozygous for the wild type at each site (n); if at any site s we have $n = Nt$ where Nt is the current number of individuals then lost type at the site

27 ⟨ not lost a type 27 ⟩ ≡

```
static bool not_lost_type ( std::vector < unsigned long > &nhomozygous, const
        std::vector < unsigned long > &population, const std::vector
        < std::pair < unsigned long , unsigned long >> &mtafla ) {
        std::fill(std::begin(nhomozygous), std::end(nhomozygous), 0);      /*
        see § 3.9 for current_number_individuals */
    const unsigned long Nt = current_number_individuals(population);
    for (unsigned long nullsite = 0; nullsite < GLOBAL_CONST_NUMBER_SITES;
          ++nullsite) {      /*
          see § 3.19 for checknullsite */
      nhomozygous[nullsite] = checknullsite(nullsite, population, mtafla);
    }
    return std::all_of (std::begin(nhomozygous), std::end(nhomozygous),
            [Nt](unsigned long n)
    {
      return n < Nt;
    }
    ) ; }
```

This code is used in chunk 30.

3.24 bottleneck

ec:bottleneck⟩

remove individuals not surviving a bottleneck using *sample_genotype_parent* in § 3.16

28 ⟨ generate a bottleneck 28 ⟩ ≡

```
static unsigned long remove_not_surviving_bottleneck ( std::vector < unsigned
    long > &v_p )
{
  unsigned long x
  { }
  ;    /*
      see § 3.9 for current_number_individuals */
  const unsigned long currentNt = current_number_individuals(v_p);
  for (unsigned long i = 0; i < currentNt - GLOBAL_CONST_BOTTLENECK; ++i) {
      /* see § 3.16 for sample_genotype_parent */
    x = sample_genotype_parent(v_p);
  }
  assert(current_number_individuals(v_p) ≡ GLOBAL_CONST_BOTTLENECK);
  return current_number_individuals(v_p);
}
```

This code is used in chunk 30.

3.25 run experiments

⟨sec:run⟩

run experiments and record excursions to fixation at all sites if any occur

29 ⟨ run experiments and record result 29 ⟩ ≡

```
static void run() {    /* define the population
    */
  std::vector < unsigned long > pop(GLOBAL_CONST_TOTAL_NUMBER_PHASED_TYPES,
        0); std::vector < unsigned long >
        nhomozygouswt(GLOBAL_CONST_NUMBER_SITES, 0); std::vector < std::pair <
        unsigned long , unsigned long >> m
  {}
  ;
  m.clear();    /*
      see § 3.3 for generate_lookup_table */
  generate_lookup_table(m); std::vector < std::pair < std::vector < unsigned long > ,
        double >> pooljuveniles
  {}
  ;
  pooljuveniles.clear(); std::vector < double > viabilityweights
  {}
  ;
  viabilityweights.clear(); std::vector < double >
      v_cdf_number_juveniles_one(GLOBAL_CONST_CUTOFF + 1, 0); std::vector
      < double > v_cdf_number_juveniles_two(GLOBAL_CONST_CUTOFF + 1, 0);    /*
      compute the CDF for sampling random number of juveniles; see § 3.12 for
      cdf_number_juveniles */
  cdf_number_juveniles(v_cdf_number_juveniles_one, GLOBAL_CONST_ALPHA_ONE);
  cdf_number_juveniles(v_cdf_number_juveniles_two, GLOBAL_CONST_ALPHA_TWO);
        std::vector < double > excursion
```

```
{}
;
unsigned long currentNt
{}
;
int trials = GLOBAL_CONST_NUMBER_EXPERIMENTS + 1;
int timi
{}
; while (--trials > 0) {      /*
    see § 3.8 for initializearray */
initializearray(pop);
timi = 0;
excursion.clear();
assert(excursion.size() < 1);      /*
    see § 3.17 for clearpooljuveniles */
clearpooljuveniles(pooljuveniles);      /*
    the population evolves until fixed for the fit type at all sites or lost a fit type at
    a site; see § 3.23 for not_lost_type */
while ((pop.back() < current_number_individuals(pop)) & not_lost_type(nhomozygouswt,
        pop, m)) {
  ++timi;      /*
        see § 3.9 for current_number_individuals */
  currentNt = current_number_individuals(pop);
  excursion.push_back(static_cast<double>(pop.back())/static_cast<double>(currentNt));
      /*
        check if a bottleneck occurs and if so remove individuals not surviving a
        bottleneck; see § 3.24 for remove_not_surviving_bottleneck */
  if (gsl_rng_uniform(rngtype) < GLOBAL_CONST_PROBABILITY_BOTTLENECK) {
```

```
        /* record the number of individuals surviving a bottleneck
         */
      currentNt = remove_not_surviving_bottleneck(pop);
    }     /* generate a new pool of juveniles; see § 3.18 for new_pool_juveniles */
    new_pool_juveniles(currentNt, pooljuveniles,
        viabilityweights, pop, (gsl_rng_uniform(rngtype) < GLOBAL_CONST_EPSILON ?
        v_cdf_number_juveniles_one : v_cdf_number_juveniles_two), m);     /*
        sort juveniles; see § 3.22 for select_juveniles_according_to_weight */
    select_juveniles_according_to_weight(viabilityweights, pop, pooljuveniles);
  }     /*
        generated one experiment; record the result; see § 3.9 for
        current_number_individuals */
  std::cout << (pop.back() < current_number_individuals(pop) ? 0 : 1) << ' ' << timi <<
      '\n'; if (pop.back() == current_number_individuals(pop)) {
    /* fixation occurs so record the excursion
     */
  std::ofstream outfile("tmpexcurs", std::ios_base::app); for (const auto &x:excursion)
  {
    outfile << x << ' ';
  }
  outfile << '\n';
  outfile.close(); } } }
```

This code is used in chunk 30.

3.26 the main module

⟨sec:main⟩

the `main` function requires a random seed; so either give a specific number or call with

```
./outfile $(shuf -i <range> -n1)
```

where `range` is a range of numbers, e.g. `4343-232383`.

30 /*

§ 3.1 */

⟨ Includes 5 ⟩ /*

§ 3.2 */

⟨ gsl random number generator 6 ⟩ /*

§ 3.3 */

⟨ generate lookup table 7 ⟩ /*

§ 3.4 */

⟨ lookup function 8 ⟩ /*

§ 3.5 */

⟨ check for recombination 9 ⟩ /*

§ 3.6 */

⟨ recombine haplotypes 10 ⟩ /*

§ 3.7 */

⟨ sample haplotype index 11 ⟩ /*

§ 3.8 */

⟨ initialize population array 12 ⟩ /*

§ 3.9 */

⟨ current number *Nt* 13 ⟩ /*

§ 3.10 */

⟨ viability weight 14 ⟩ /*

§ 3.14 */

⟨ addjuv 18 ⟩ /*

§ 3.11 */

⟨ kernel 15 ⟩ /*

§ 3.12 */

⟨ cdf 16 ⟩ /*

§ 3.13 */

⟨ sample litter size 17 ⟩ /*

§ 3.15 */

⟨ add sibship 19 ⟩ /*

§ 3.16 */

⟨ onehypergeometric 20 ⟩ /*

§ 3.17 */

⟨ clear the pool 21 ⟩ /*

§ 3.18 */

⟨ a new pool 22 ⟩ /*

§ 3.19 */

⟨ count homozygous at site 23 ⟩ /*

§ 3.20 */

⟨ compare 24 ⟩ /*

§ 3.21 */

⟨ *n*th element 25 ⟩ /*

§ 3.22 */

⟨ sorting 26 ⟩ /*

§ 3.23 */

⟨ not lost a type 27 ⟩ /*

§ 3.24 */

⟨ generate a bottleneck 28 ⟩ /*

§ 3.25 */

⟨ run experiments and record result 29 ⟩

int *main*(**int** *argc*, **char** ∗*argv*[ ])

```
{       /* initialise the GSL random number generator § 3.2
         */
  setup_rng(static_cast<unsigned long>(atoi(argv[1])));
        /* free the GSL random number generator
         */
  gsl_rng_free(rngtype);
  return GSL_SUCCESS;
}
```

Index

List of Refinements