Fixation at many sites

Bjarki Eldon[1] [2]

Abstract

This code generates excursions of the evolution of a diploid population partitioned into an arbitrary number of unlinked sites and two genetic types at each site, with viability weight determined by $W = e^{-sf(g)}$, where $g = (g_1, \ldots, g_L)$ are the genotypes of a given individual at L sites, f is a function determining how the genotypes affect the trait value, and $s > 0$ is the strength of selection on the trait. The population evolves according to a model of random sweepstakes and randomly occurring bottlenecks and viability selection. The code can be used to estimate the probability of fixation of the (L-site) type conferring advantage, and the expected time to fixation conditional on fixation of the type conferring advantage.

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, Wolfgang Stephan, and BE. BE also acknowledges Start-up module grants through SPP 1819 with Jere Koskela and Maite Wilke-Berenguer, and with Iulia Dahmer.
June 28, 2022

1 Copyright

2 introduction

We are interested in how random sweepstakes, randomly occurring bottlenecks, and dominance mechanisms affect the probability of fixation, and the time to fixation conditional on fixation of the type conferring selective advantage. We consider a diploid population partitioned into an arbitrary number of unlinked sites and two types at each site. The population is evolving according to random sweepstakes and randomly occurring bottlenecks. In between bottlenecks the population experiences viability selection provided the number of juveniles produced each time exceeds the given carrying capacity.

Consider a diploid population starting with $2N$ diploid individuals. Let $X, X_1^N, \ldots, X_N^N$ be i.i.d. discrete random variables taking values in $\{2, \ldots, u(N)\}$; the $X_1^N, \ldots, X_M^N$ denote the random number of juveniles independently produced in a given generation by M parent pairs according to

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

The mass in Eq (1) is normalised so that $\mathbb{P}(1 \le X \le u(N)) = 1$, and $\mathbb{P}(X = k) \ge \mathbb{P}(X = k+1)$. Given a pool of at least $2M$ juveniles, we sample $2M$ juveniles for the next generation. Leaving out an atom at zero and one gives $X_1^N + \cdots + X_N^N \ge 2M$ almost surely, guaranteeing that we always have at least $2M$ juveniles to choose from in each generation. We randomize on α so that with probability $1 - \varepsilon_N$ all parent pairs produce juveniles according to Eq (1) with $\alpha = \alpha_2 \ge 2$, and with $\alpha = \alpha_1 \in (1, 2)$ with probability ε_N.

We admit randomly occurring bottlenecks; in the beginning of every generation a bottleneck occurs with a fixed probability, and when it happens we sample uniformly at random and without replacement a fixed number of diploid individuals to survive the bottleneck. The surviving individuals then continue to produce juveniles. If the total number of juveniles does not exceed the carrying capacity (C) all the juveniles survive, otherwise we sample C of them without replacement and with weights. Let A denote the event a bottleneck occurs, the population size at time N_{t+1} is then given by $N_{t+1} = \mathbb{1}_{\{S_{\lfloor M/2 \rfloor} \le C\}} S_{\lfloor M/2 \rfloor} + \mathbb{1}_{\{S_{\lfloor M/2 \rfloor} > C\}} C$

where $S_{\lfloor M/2 \rfloor}$ is the total number of juveniles produced by $\lfloor M/2 \rfloor$ parent pairs, and $M = \mathbb{1}_{\{A\}} B + \mathbb{1}_{\{A^{\complement}\}} N_t$ where B is the bottleneck 'size', i.e. the number of individuals surviving a bottleneck. Viability selection therefore does not kick in unless the number of juveniles at any time exceeds the carrying capacity; given recurrent bottlenecks the population may be evolving neutrally for several generations.

The code can be used to estimate the probability of fixation at all sites for the type conferring advantage, and the time to fixation conditional on fixation. In this context 'fixation' means fixation at all sites of the type conferring advantage; if the type is lost at any one site fixation cannot occur.

At the time of writing weight of a juvenile required for viability selection is taken as a random exponential with rate $\exp(-sf(g))$ where $s > 0$ is the strength of selection and $g = (g_1, \ldots, g_L)$ are the genotypes at the L sites (§ 3.16, § 3.17). One can assume various dominance mechanisms, and different dominance mechanisms for different sites.

3 code

The R code § 3.1 produces all possible L-site types, for a given L there's 3^L of them. For two sites ($L = 2$) with nine possible types the matrix is

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

Sections § 3.2–§ 3.30 describe the individual modules. The algorithm is formulated in the following pseudocode. Let $Y(t) := (Y_1(t), \ldots, Y_L(t)) \in [0,1]^L$ be the L-site type frequency process, where $Y_\ell(t)$ is the number of copies of the type conferring selective advantage at site ℓ relative to the population size at time t. Write $[n] := \{1, 2, \ldots, n\}$ for $n \in$

3.1 R code for types

```
# given L sites there are 3**L possible L-site types
g <- function(l, LL)
{
x <- numeric()
while( length(x) < (3**LL) ){
x <- c(x,  rep(0,3**(LL-l)),  rep(1, 3**(LL-l)), rep(2, 3**(LL-l))) }
return (x)
}
# example:  set the number of sites w to five
w <- 5
# define the matrix for the types
m <- matrix( 0, 3**w, w)
# compute the  matrix of types at w  sites
for(l in 1:w){
m[, l]  <- t(g(l, w))}
# write the matrix to a file
write(t(m), "types.txt", ncolumns =  w)
```

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

This code is used in chunk 34.

3.3 GSL random number generator

initialise a gsl random number generator; called in § 3.31

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 34.

3.4 look up an index

Compute an index given an L-site type $(i_1, \ldots, i_L) \in \{0, 1, 2\}^L$ as

$$f((i_1, \ldots, i_L)) = \sum_{\ell=1}^{L} i_\ell 3^{L-\ell}. \tag{2}$$

This enables us to look up the number of individuals in the population with the given L-site type

7 ⟨ lookup 7 ⟩ ≡

```
static size_t lookup ( const std::vector < short > &types )
{
  size_t n = 0;
  for (int ell = 1; ell ≤ GLOBAL_NUMBER_SITES; ++ell) {
    n += types[ell-1]*static_cast⟨short⟩(gsl_sf_pow_int(3, GLOBAL_NUMBER_SITES-ell));
  }
  return (n);
}
```

This code is used in chunk 34.

3.5 replicate a value

replicate a given value a given number of times as in R function `rep`; required for enumerating all the possible L-site types

8 ⟨ as R rep 8 ⟩ ≡

```
static void crep (const double LL, const double l, const short n, std::vector <
    short > &y )
{
  y.clear( );
  y.resize(pow(3, LL − l));
  y.assign(pow(3, LL − l), n);
}
```

This code is used in chunk 34.

3.6 one column of type array

write out one column of the type array, i.e. the expanded types at one site; required for producing all he possible L-site types

9 ⟨ L-site type array one column 9 ⟩ ≡

```
static void g (const double LL, const double l, std::vector < short > &x ) { x.clear( );
        std::vector < short > tmp
{ }
;
const size_t m = static_cast⟨size_t⟩(pow(3., LL));
while (x.size( ) < m) {
  for (short i = 0; i < 3; ++i) {      /*
        § 3.5 */
    crep(LL, l, i, tmp);
    x.insert(x.end( ), tmp.begin( ), tmp.end( ));
  }
}
}
```

This code is used in chunk 34.

3.7 write array of L-site types

write the array of all L-site types

10 ⟨ all L-site types 10 ⟩ ≡

```
static void writearray(const double LLw, gsl_matrix_short * M) { std::vector < short > d
    {}
    ;
    for (size_t k = 1; k ≤ static_cast⟨size_t⟩(LLw); ++k) {      /*
            § 3.6 */
      g(LLw, k, d);      /*
            copy into array */
      for (size_t u = 0; u < d.size(); ++u) {
        gsl_matrix_short_set(M, u, k − 1, d[u]);
      }
    }
    }
```

This code is used in chunk 34.

3.8 initialize containers

initialize the main containers; we start with $2N - L$ individuals homozygous for the wild type at all sites, and L distinct individuals each heterozygous at a distinct site.

11 ⟨ container initialization 11 ⟩ ≡

```
static void init_containers ( std::vector < unsigned > &population, std::vector <
    double > &cdf_one, std::vector < double > &cdf_two ) { population.clear( );
population.assign(GLOBAL_NUMBER_TYPES, 0);      /*
    set 2N − L individuals homozygous for the wild type at all sites */
population[0] = GLOBAL_CONST_II − GLOBAL_NUMBER_SITES;      /*
    set an L site type heterozygous at one site and homozygous for wild type at all other
    sites */
std::vector < short > types(GLOBAL_NUMBER_SITES, 0);
for (unsigned i = 0; i < GLOBAL_NUMBER_SITES; ++i) {
  std::fill(types.begin( ), types.end( ), 0);
  types[i] = 1;      /*
      set the number of individuals heterozygous at site i to one § 3.4 */
  population[lookup(types)] = 1;
}      /*
      initialize the containers for the CDFs for the number of juveniles */
cdf_one.clear( );
cdf_two.clear( );
cdf_one.reserve(GLOBAL_CONST_CUTOFF_ONE + 2);
cdf_two.reserve(GLOBAL_CONST_CUTOFF_TWO + 2);
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);
```

$assert(cdf\_two.size() \equiv 2);$ }

This code is used in chunk 34.

3.9 initialize for an excursion

initialize the population array for a new trajectory

12 ⟨ newtrajectory 12 ⟩ ≡

```
static void init_for_trajectory ( std::vector < unsigned > &population ) {
    std::fill(population.begin( ), population.end( ), 0);      /*
    set 2N − L individuals as homozygous for wild type at all sites */
population[0] = GLOBAL_CONST_II − GLOBAL_NUMBER_SITES; std::vector <
    short > types(GLOBAL_NUMBER_SITES, 0);
for (unsigned i = 0; i < GLOBAL_NUMBER_SITES; ++i) {
  std::fill(types.begin( ), types.end( ), 0);      /*
      set one individual as heterozygous at site i and homozygous for wild type at all
      other sites */
  types[i] = 1;
  population[lookup(types)] = 1;
}
}
```

This code is used in chunk 34.

3.10 total number of individuals in population

return the current total number of individuals in the population

13 ⟨ Nt 13 ⟩ ≡

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

This code is used in chunk 34.

3.11 sample index of parent

sample index of L-site type of parent without replacement and update the number of remaining individuals accordingly

14 ⟨ getparentindex 14 ⟩ ≡

```
static int sample_genotype_parent ( std::vector < unsigned > &p, gsl_rng*r )
{      /*
         p is population */
  int i = 0;
  unsigned int nothers = current_number_individuals(p) - p[i];
  unsigned int x = gsl_ran_hypergeometric(r, p[0], nothers, 1);
  while ((x < 1) ∧ (i < GLOBAL_NUMBER_TYPES)) {
    ++i;
    nothers -= p[i];
    x = gsl_ran_hypergeometric(r, 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 index of the genotype of the parent */      /*
       index is between 0 and GLOBAL_NUMBER_TYPES */
  return i;
}
```

This code is used in chunk 34.

3.12 all juveniles survive

all juveniles survive

15 ⟨ all juveniles 15 ⟩ ≡

```
static void update_population_all_juveniles ( const std::vector < std::pair < size_t ,
    double >> &juveniles, std::vector < unsigned > &population ) {      /*
    set number of all types to zero */
std::fill(population.begin( ), population.end( ), 0);      /*
    § 3.10 */
assert(current_number_individuals(population) < 1); for (const auto &j:juveniles)
{      /*
      j[0] is type index of juvenile j */
  population[std::get < 0 > (j)] += 1;
}
}
```

This code is used in chunk 34.

3.13 sample a juvenile type

assign single site genotype to juvenile given genotypes in parents following Mendel's laws; in our coding 0 corresponds to homozygous 0/0 for the wild type, 1 corresponds to the heterozygote, and 2 to the homozygote for the mutation

16 ⟨ sample single site type 16 ⟩ ≡

```
static short assign_type_one_site(const short gone, const short gtwo, gsl_rng * r)
{     /*
        gone and gtwo are the two parent types */
  short 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;
  }
  assert(g ≡ 0 ∨ g ≡ 1 ∨ g ≡ 2);
  return g;
}
```

This code is used in chunk 34.

3.14 read in file with types

read in file with all types for given number of sites produced by the R code in § 3.1

17 ⟨ file of types 17 ⟩ ≡

```
static void read_types ( const std::vector < unsigned > &p, gsl_matrix_short*M )
{
  std::ifstream f("types_five_sites.txt");
  short x
  {}
  ;
  for (int i = 0; i < GLOBAL_NUMBER_TYPES; ++i) {
    for (int j = 0; j < GLOBAL_NUMBER_SITES; ++j) {
      f >> x;
      gsl_matrix_short_set(M, i, j, x);
    }
  }
  f.close( );      /*
      print matrix for check */
  int z = 0;
  for (int i = 0; i < GLOBAL_NUMBER_TYPES; ++i) {
    for (int j = 0; j < GLOBAL_NUMBER_SITES; ++j) {
      std::cout << gsl_matrix_short_get(M, i, j) << ' ';
    }
    std::cout << p[z] << '\n';
    ++z;
  }
}
```

This code is used in chunk 34.

3.15 a random number of juveniles

sample a random number of juveniles with a distribution based on Eq (1),
returning $\min\{j \in \mathbb{N} : F(j) \geq u\}$ where u is a given random uniform and F the CDF

18 ⟨ randomnumberjuvs 18 ⟩ ≡

```
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 34.

3.16 weight of a site

compute the contribution of a site to the weight

19 ⟨ weight 19 ⟩ ≡

```
static double weight(const short x)
{
  return ((x < 1 ? 2. : 0.)/GLOBAL_NUMBER_SITES_d);
}
```

This code is used in chunk 34.

3.17 compute weight

compute weight of a juvenile given the type at all sites

20 ⟨ compute the weight 20 ⟩ ≡

```
static double computeweight ( const std::vector < short > &types, gsl_rng*r ) {      /*
    types is the vector of type at each site of juvenile */
double g = 0;      /*
    compute the average contribution over the sites */
for (const auto &s:types)
{     /* compute the contribution of site s to the weight § 3.16 */
  g += weight(s);
}     /*
      return the weight as a random exponential with rate exp(−sg^2) where g is from
      § 3.16 */
return (gsl_ran_exponential(r,1./exp((-GLOBAL_CONST_SELECTION)*pow(g,2.)))); }
```

This code is used in chunk 34.

3.18 add one juvenile

add one juvenile to the pool of juveniles

21 ⟨ add one juvenile 21 ⟩ ≡

```
static void add_juvenile (const int type_index_one,
        const int type_index_two, gsl_matrix_short * Mtypes, std::vector < std::pair <
        size_t , double >> &jvs, gsl_rng*r ) { std::vector < short > types_juvenile
{ }
;
types_juvenile.clear( );      /*
    get the types of the juvenile */
for (int s = 0; s < GLOBAL_NUMBER_SITES; ++s) {      /*
        sample and record the type at site s § 3.13 */
  types_juvenile.push_back(assign_type_one_site(gsl_matrix_short_get(Mtypes, type_index_one,
      s), gsl_matrix_short_get(Mtypes, type_index_two, s), r));
}     /*
        compute the weight § 3.17 and type index § 3.4 and add the juvenile to the vector
        of juves */
jvs.push_back(std::make_pair(lookup(types_juvenile), computeweight(types_juvenile, r))); }
```

This code is used in chunk 34.

3.19 juveniles for a parent pair

produce a set of juveniles for one parent pair with given type indexes

22 ⟨ produce juvs for parent pair 22 ⟩ ≡

```
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_matrix_short*Mtypes, gsl_rng*r )
{   /*
      gone and gtwo are the type indexes for the two parents; Mtypes
      the matrix of types */   /*
      first sample a random number of juveniles § 3.15 */
  const size_t numberj = sample_random_number_juveniles(conetwo,
      cdfone, cdftwo, r);
  assert(numberj > 1);   /*
      add the sampled number of juveniles to the pool § 3.18 */
  for (size_t j = 0; j < numberj; ++j) {
    add_juvenile(gone, gtwo, Mtypes, jvs, r);
  }
}
```

This code is used in chunk 34.

3.20 a new pool of juveniles

generate a new pool of juveniles

23 ⟨ new pool juvs 23 ⟩ ≡

```
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_matrix_short*Mtypes, gsl_rng*r )
{
  jvs.clear( );
  jvs.shrink_to_fit( );
  assert(jvs.size( ) < 1);
  int gone
  { }
  ;
  int gtwo
  { }
  ;   /*
      sample distribution of number of juveniles */
  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 § 3.10 */
  const double currenti = current_number_individuals(p);
  assert(currenti < GLOBAL_CONST_I + 1);
  for (double i = 0; i < floor(currenti/2.); ++i) {   /*
        § 3.11 */
    gone = sample_genotype_parent(p, r);
    gtwo = sample_genotype_parent(p, r);   /*
        gone and gtwo are the type indexes of the two parents */
```

```
        assert(gone > -1);
        assert(gtwo > -1);      /*
            § 3.19 */
        add_juveniles_for_given_parent_pair(cdfone, cdftwo, jvs, gone, gtwo, conetwo,
            Mtypes, r);
      }
      assert(jvs.size( ) ≥ static_cast⟨size_t⟩(currenti));
    }
```

This code is used in chunk 34.

3.21 surviving a bottleneck

sample diploid individuals surviving a bottleneck; we sample uniformly at random without replacement

24 ⟨ survive bottleneck 24 ⟩ ≡

```
static void sample_surviving_bottleneck ( std::vector < unsigned > &p, gsl_rng*r )
{
  int i = 0;
  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 < GLOBAL_NUMBER_TYPES - 2) {
    ++i;
    nothers -= p[i];
    newn = (remaining > 0 ? gsl_ran_hypergeometric(r, p[i], nothers, remaining) : 0);
    p[i] = newn;
    remaining -= newn;
  }
  assert(GLOBAL_NUMBER_TYPES - 1 < p.size( ));
  p[GLOBAL_NUMBER_TYPES - 1] = (remaining < GLOBAL_CONST_BOTTLENECK ?
      remaining : GLOBAL_CONST_II);
  assert(current_number_individuals(p) ≥ GLOBAL_CONST_BOTTLENECK);
}
```

This code is used in chunk 34.

3.22 check if lost a type

return True if not lost the type conferring advantage at any site, otherwise False

25 ⟨ mutation is still around 25 ⟩ ≡

```
static bool not_lost_type ( const std::vector < unsigned > &p, gsl_matrix_short*M ) {
    std::vector < unsigned > x(GLOBAL_NUMBER_SITES, 0);
for (int i = 0; i < GLOBAL_NUMBER_TYPES; ++i) {
  for (size_t s = 0; s < GLOBAL_NUMBER_SITES; ++s) {
    x[s] += gsl_matrix_short_get(M, i, s) < 1 ? p[i] : 0;
  }
}    /*
       GLOBAL_CONST_II is 2N the maximum number of diploid individuals */
return std::all_of (x.begin( ), x.end( ), [](unsigned n)
{
  return n < GLOBAL_CONST_II;
}
) ; }
```

This code is used in chunk 34.

3.23 sample juveniles by weight

sample juveniles by weight; the surviving juveniles in number equal the carrying capacity

26 ⟨ pick by weight 26 ⟩ ≡

```
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.);
  std::fill(population.begin( ), population.end( ), 0);      /* § 3.10 */
  assert(current_number_individuals(population) < 1);      /*
      check number of juveniles and nth element */
  assert(juveniles.size( ) ≥ GLOBAL_CONST_II);
  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;
  }     /* § 3.10 */
  assert(current_number_individuals(population) ≡ GLOBAL_CONST_II);
}
```

This code is used in chunk 34.

3.24 compare two juveniles

compare the weight of two juveniles, needed for computing the $2N$th smallest weight among the weight of juveniles

27 ⟨ compare 27 ⟩ ≡

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

This code is used in chunk 34.

3.25 computing the $2N$th smallest weight

compute the $2N$th smallest weight among the weight of juveniles, needed for sampling the juveniles according to weight when the total number of juveniles exceeds the carrying capacity

28 ⟨ nth 28 ⟩ ≡

```
static double nthelm ( std::vector < std::pair < size_t , double >> &juveniles )
{       /* § 3.24 */
  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 34.

3.26 take one step

step through one generation

29 ⟨ one step 29 ⟩ ≡

```
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_matrix_short*M, gsl_rng*r )
{
  double nth
  {}
  ;     /*
      check if bottleneck */
  if (gsl_rng_uniform(r) < GLOBAL_CONST_PROBABILITY_BOTTLENECK) {
      /*
       bottleneck occurs ; sample surviving types and update population
       § 3.21 */
    sample_surviving_bottleneck(p, r);
  }     /*
       first check if lost type at any site § 3.22 */
  if (not_lost_type(p, M)) {     /*
       not lost type; check if fixed at all sites */
    if (p.back( ) < GLOBAL_CONST_II) {     /*
         not all individuals of type 2 at all sites, so sample juveniles § 3.20 */
      generate_pool_juveniles(jvs, p, cdfone, cdftwo, M, r);
      if (jvs.size( ) ≤ GLOBAL_CONST_II) {     /*
           total number of juveniles not over capacity so all survive § 3.12 */
        update_population_all_juveniles(jvs, p);     /*
            § 3.10 */
        assert(current_number_individuals(p) ≥ GLOBAL_CONST_BOTTLENECK);
```

```
        }
        else {      /*
                need to sort juveniles and sample according to weight § 3.25 */
          nth = nthelm(jvs);      /*
                § 3.23 */
          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 34.

3.27 generate one excursion

generate one excursion and record the result

30 ⟨ one excursion 30 ⟩ ≡

```
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_matrix_short*M, gsl_rng*r ) { init_for_trajectory(p);
    const std::string skra = "excursions_sites_" + std::to_string(numer);
        std::vector < double > excursion_to_fixation
    { }
    ;
    int timi = 0;
    while ((not_lost_type(p, M)) ∧ (p.back( ) < GLOBAL_CONST_II)) {      /*
        record the number of diploid individuals homozygous 1/1 at all
        sites over current number of diploid individuals */
      assert(GLOBAL_CONST_PROBABILITY_BOTTLENECK > 0. ?
        (current_number_individuals(p) ≥ GLOBAL_CONST_BOTTLENECK) :
        (1 > 0));
      excursion_to_fixation.push_back(static_cast⟨double⟩(p.back( ))/static_cast⟨double⟩(
      ++timi;
      onestep(p, cdfone, cdftwo, jvs, M, r);
    }
    if (p.back( ) ≡ GLOBAL_CONST_II) {      /*
        fixation occurs so print excursion to file */
    std::ofstream outfile(skra, std::ios_base::app); for (const auto
        &y:excursion_to_fixation)
    {
      outfile ≪ y ≪ '␣';
```

```
}
outfile ≪ '\n';
outfile.close( ); } std::cout ≪ (p[GLOBAL_NUMBER_TYPES − 1] <
    GLOBAL_CONST_II ? 0 : 1) ≪ '␣' ≪ timi ≪ '\n'; }
```

This code is used in chunk 34.

3.28 mass function for number of juveniles

the mass function for number of juveniles as in Eq (1)

31 ⟨ mass 31 ⟩ ≡

```
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 34.

3.29 initialise the CDF

initialise the CDF corresponding to Eq (1) for the distribution of the number of juveniles

32 ⟨ init cdf 32 ⟩ ≡

```
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 34.

3.30 run a given number of experiments

run a given number of experiments and record the results

33 ⟨ generate many excursions 33 ⟩ ≡

```
static void runsims(const int x, gsl_rng * r) { std::vector < unsigned
        int > population(GLOBAL_NUMBER_TYPES, 0); std::vector < std::pair <
        size_t , double ≫ juveniles
{}
; std::vector < double > cdf_one
{}
; std::vector < double > cdf_two
{}
;    /* § 3.8 */
init_containers(population, cdf_one, cdf_two);    /* § 3.29 */
initialise_cdf (cdf_one, cdf_two);    /*
    write the matrix of all L-site types § 3.7 */
gsl_matrix_short * M = gsl_matrix_short_calloc(GLOBAL_NUMBER_TYPES,
    GLOBAL_NUMBER_SITES);
writearray(static_cast⟨double⟩(GLOBAL_NUMBER_SITES), M);
int z = GLOBAL_CONST_NUMBER_EXPERIMENTS + 1;
while (−−z > 0) {    /*
      § 3.27 */
  trajectory(population, cdf_one, cdf_two, juveniles, x, M, r);
}
gsl_matrix_short_free(M); }
```

This code is used in chunk 34.

3.31 main module

the `main` function calling

```
34      /*
        §3.2 */
    ⟨ Includes 5 ⟩    /*
        § 3.3 */
    ⟨ gsl random number generator 6 ⟩    /*
        § 3.4 */
    ⟨ lookup 7 ⟩    /*
        § 3.8 */
    ⟨ container initialization 11 ⟩    /*
        § 3.9 */
    ⟨ newtrajectory 12 ⟩    /*
        § 3.10 */
    ⟨ Nt 13 ⟩    /*
        § 3.11 */
    ⟨ getparentindex 14 ⟩    /*
        § 3.12 */
    ⟨ all juveniles 15 ⟩    /*
        § 3.13 */
    ⟨ sample single site type 16 ⟩    /*
        § 3.14 */
    ⟨ file of types 17 ⟩    /*
        § 3.15 */
    ⟨ randomnumberjuvs 18 ⟩    /*
        § 3.16 */
    ⟨ weight 19 ⟩    /*
        § 3.17 */
```

⟨ compute the weight 20 ⟩ /*
§ 3.18 */
⟨ add one juvenile 21 ⟩ /*
§ 3.19 */
⟨ produce juvs for parent pair 22 ⟩ /*
§ 3.20 */
⟨ new pool juvs 23 ⟩ /*
§ 3.21 */
⟨ survive bottleneck 24 ⟩ /*
§ 3.22 */
⟨ mutation is still around 25 ⟩ /*
§ 3.23 */
⟨ pick by weight 26 ⟩ /*
§ 3.24 */
⟨ compare 27 ⟩ /*
§ 3.25 */
⟨ nth 28 ⟩ /*
§ 3.26 */
⟨ one step 29 ⟩ /*
§ 3.27 */
⟨ one excursion 30 ⟩ /*
§ 3.28 */
⟨ mass 31 ⟩ /*
§ 3.29 */
⟨ init cdf 32 ⟩
⟨ as R rep 8 ⟩ /*
§ 3.6 */
⟨ L-site type array one column 9 ⟩ /*

```
    § 3.7 */
⟨ all L-site types 10 ⟩    /*
    § 3.30 */
⟨ generate many excursions 33 ⟩    /*
    § 3.5 */
int main(int argc, char *argv[ ])
    {    /*
            § 3.3 */
      setup_rng(static_cast⟨unsigned long⟩(atoi(argv[1])));    /*
            § 3.30 */
      runsims(atoi(argv[1]), rngtype);
      gsl_rng_free(rngtype);
      return GSL_SUCCESS;
    }
```

Index

List of Refinements

⟨ Includes 5 ⟩ Used in chunk 34.

⟨ L-site type array one column 9 ⟩ Used in chunk 34.

⟨ Nt 13 ⟩ Used in chunk 34.

⟨ add one juvenile 21 ⟩ Used in chunk 34.

⟨ all L-site types 10 ⟩ Used in chunk 34.

⟨ all juveniles 15 ⟩ Used in chunk 34.

⟨ as R rep 8 ⟩ Used in chunk 34.

⟨ compare 27 ⟩ Used in chunk 34.

⟨ compute the weight 20 ⟩ Used in chunk 34.

⟨ container initialization 11 ⟩ Used in chunk 34.

⟨ file of types 17 ⟩ Used in chunk 34.

⟨ generate many excursions 33 ⟩ Used in chunk 34.

⟨ getparentindex 14 ⟩ Used in chunk 34.

⟨ gsl random number generator 6 ⟩ Used in chunk 34.

⟨ init cdf 32 ⟩ Used in chunk 34.

⟨ lookup 7 ⟩ Used in chunk 34.

⟨ mass 31 ⟩ Used in chunk 34.

⟨ mutation is still around 25 ⟩ Used in chunk 34.

⟨ new pool juvs 23 ⟩ Used in chunk 34.

⟨ newtrajectory 12 ⟩ Used in chunk 34.

⟨ nth 28 ⟩ Used in chunk 34.

⟨ one excursion 30 ⟩ Used in chunk 34.

⟨ one step 29 ⟩ Used in chunk 34.

⟨ pick by weight 26 ⟩ Used in chunk 34.

⟨ produce juvs for parent pair 22 ⟩ Used in chunk 34.

⟨ randomnumberjuvs 18 ⟩ Used in chunk 34.

⟨ sample single site type 16 ⟩ Used in chunk 34.

⟨ survive bottleneck 24 ⟩ Used in chunk 34.

⟨ weight 19 ⟩ Used in chunk 34.